ASYNCHRONOUS HARDWARE-ENFORCED MEMORY SAFETY

BY

KUAN-YU TSENG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Advisers:

Professor Ravishankar K. Iyer
Professor Zbigniew T. Kalbarczyk

ABSTRACT

Memory corruption attacks, such as buffer overflow attacks, have been threatening software security for more than three decades. Despite tremendous efforts by developers and researchers to prevent programs written in memory-unsafe language (e.g., C/C++) from memory errors (the root cause of memory corruption attacks), the drawbacks and limitations of prior protection mechanisms impede their wide deployment. Prior approaches suffer from either (1) incomplete coverage of memory errors, (2) prohibitively high runtime overhead, (3) weak protection for metadata used by the approach, (4) low source compatibility to legacy code, (5) low binary compatibility to compiled binaries, (6) limited modularity support, or (7) low scalability for larger programs.

This thesis presents AHEMS, an architectural support that ensures both *spatial* and *temporal* memory safety based on the pointer-based approach to overcome the aforementioned drawbacks. AHEMS provides three novel features that allow for fast, flexible, and secure memory safety checking. First, AHEMS checks memory safety *asynchronously* to the main processor so that the runtime overhead is very low. Second, AHEMS can be flexibly implemented either in a processor, as a co-processor, or as an external device, depending on the designer's choice. Third, AHEMS provides physical isolation for the metadata (i.e., base and bounds information) so that the metadata cannot be tampered with by any means.

We implement an FPGA prototype for AHEMS that allows us to evaluate its detection coverage, runtime overhead, critical path, hardware overhead, and power consumption. Our experiment shows that AHEMS passes 676 security test cases out of 11 different CWEs (including spatial and temporal memory errors) and only incurs as little as 10.6% runtime overhead with a negligible impact on the critical path (0.06% overhead) and power consumption (0.5% overhead).

To my family and my friends, for their love and support.

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my advisers, Prof. Zbigniew Kalbarczyk and Prof. Ravishankar Iyer, for their patient guidance and valuable advice during my two years as a graduate student in UIUC. The thought-provoking discussions and big pictures I got from them were extremely helpful to the completion of this thesis. I would like to thank Prof. Deming Chen for his generosity to lend me the FPGA board for my experiments. I also appreciate Daniel Chen, who first brought me into the DEPEND group and helped me start my research. I am indebted to my groupmates: Valentin Sidea for the hardware knowledge he imparted and the alpha version of AHEMS he built, Zachary Estrada for his English suggestions, and Dao Lu for his tremendous help on the implementation of AHEMS. Many thanks also to my other groupmates – Phuong Cao, Cuong Pham, Hui Lin, Skylar Lee, Homa Alemzadeh, Catello Di Martino, and Keun Soo Kim – for their insightful feedback on my research and our enjoyable group retreats.

I would like to thank my friends in UIUC – Po-Liang Wu, Kai-Wei Chang, Chi-Yao Hung, Chia-Chi Lin, Lue-Jane Lee, Chen-Tse Tsai, Kyle Jao, Victor Lu, Li-Lun Wang, Shih-Wen Huang, Pei-Fen Tu, Chih-Chieh Yang, Chen-Hsuan Lin, Chang-Tse Hsieh, Wei-Fen Chen, Judy Hsu, Christine Tseng, TK Wei, Shu-Han Chao, Jonathan Wang, Jimmy Chu, Frank Pan, Amy Chu, and Melanie Hsu – for their company, encouragement, and cameraderie. I would also like to thank my roommates – Fardin Abdi, Nikita Spirin, and Jason Croft – for their support and the nonstop discussions we had.

Last, I would like to express my deep gratitude to my brother, Chun-Wei Tseng, and my parents, Ming-Cheng Tseng and Pi-Ju Wang. My brother encouraged me to pursue a master's degree in the US, which led me to this rewarding journey. My parents, although in Taiwan and far from the US, stood by me with their love, support, and company.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

C is the most popular programming language to date, as reported by TIOBE [1] and Programming Language Popularity [2]. It is widely used by many applications, especially system programs or performance-critical applications, for several reasons. (1) C has good support for low-level controls on the underlying data representation, which is required by many system programs. (2) C's manual memory management allows efficient reuse of memory when memory is limited. (3) C runs faster than many other languages because manual memory management also eliminates the need for the additional, program-slowing processing done by other languages such as garbage collection or dynamic typing. However, these advantages come with a cost. In particular, C is memory-unsafe, which means that a pointer in C may point to an object that no longer exists or that is not the *intended referent*[1] of the pointer. This means the dereference of the pointer may access a memory location not intended by the programmer. This deficiency leads to a number of memory errors, such as buffer overflows, use-after-free, or invalid frees, which can be exploited to construct memory corruption attacks.

Memory corruption attacks have existed for more than three decades, since the first public discussion of memory errors in 1972 by James Anderson [3]. However, it is still ranked among the top 3 of the CWE/SANS top-25 most dangerous software errors [4] in 2011. Memory errors can be categorized into *spatial memory errors* or *temporal memory errors. Spatial memory errors* occur when a pointer is used to access the memory beyond the base and bound of its intended referent (e.g., buffer overflows). *Temporal memory errors* occur when a pointer is used to access an object that no longer exists (e.g., use-after-free errors, dangling-pointer errors, or double-free errors).

Although a plethora of countermeasures are proposed to protect different

[1] the intended referent of a pointer is the object to which the pointer is supposed to point according to the semantics of C source code

aspects of systems from memory corruption attacks, most of them are not comprehensive enough to defeat all such attacks. For example, protection schemes that enforce code integrity [5–9] only prevent code-injection attacks. Approaches that protect the program stack [5–12] or program heap [13–19] from tampering do not prevent attacks that allow arbitrary memory writes. Techniques that embrace the power of randomization to protect code or data memory space [20–28] are mostly subject to information leakage attacks that leak the keys or the layout after randomization; thus they can only be used to enhance security, not guarantee it. Methods that employ static analysis [29–34] or dynamic analysis [35–42] to ensure integer integrity can only hamper integer-overflow attacks. Defenses that guarantee control-flow integrity [43–54] can prevent many memory corruption attacks. Unfortunately, they cannot detect non-control-data attacks that do not divert the legal control flow. Mechanisms that ensure information-flow integrity [55–72] suffer from numerous false positives and false negatives, which makes them impractical in real-world systems. Finally, data-flow integrity [73] prohibits many memory corruption attacks but misses some attacks due to the imprecision of the data-flow analysis. The above-mentioned approaches all protect against a subset of memory corruption attacks and may be orthogonal to one another to form complete coverage of memory corruption attacks. However, the combination of different approaches may be non-trivial, so research moves toward eliminating the memory corruption attacks at their root cause: memory errors.

Approaches aimed to eliminate all memory errors (1) check the bounds information to make sure no access violates spatial memory safety and (2) check the lifetime of an object to ensure no access violates temporal memory safety. There are two main approaches to memory safety: *object-based* approaches [74–86], which associate base and bounds information with each object, and *pointer-based* approaches [87–98], which associate base and bound information with each pointer. However, software implementation of these approaches primarily suffer from various problems: (1) prohibitively high runtime overhead, (2) limited binary compatibility (i.e., a protected code cannot properly interoperate with unprotected binaries/libraries whose source code is unavailable), (3) weak metadata protection, (4) incomplete memory safety, (5) limited source compatibility (i.e., manual modification to the source code is needed) (6) limited support for modularity (i.e., components cannot be handled separately), or (7) limited scalability (i.e., the approach does not scale up to large programs). These deficiencies, especially the high runtime overhead and limited binary compatibility, deter the deployment of these approaches on real-world applications. Some hardware approaches [93, 94, 97–101] further improve the runtime overhead of software approaches. Nonetheless, those hardware approaches introduce other issues to the system. For example, the state-of-the-art approach Watchdog [94], for example, needs to quintuple the size of the register file in the processor to enforce complete memory safety. The approach of Chuang et al. [98] loses binary-compatibility due to the use of fat pointers. SafeMem [99], MemTracker [100, 101], and Clause et al. [95, 96] sacrifice their detection coverage for performance. Finally, SafeProc [97] is not source-compatible.

In this thesis, we introduce AHEMS (Asynchronous Hardware-Enforced Memory Safety), which implements a pointer-based approach in hardware with *asynchronous checking* to overcome the above-mentioned deficiencies. *Asynchronous checking* allows AHEMS to offload the bounds checking operations occurring on each pointer dereference to the *security engine*, a customized hardware that can be implemented as an on-chip extension, co-processor, or external device such as a PCI card (See Chapter 5). The *security engine* can perform bounds checking asynchronously and with faster frequency than the main processor (the processor that runs the program), greatly improving the runtime overhead. Also, since the *security engine* can be decoupled from the chip with the main processor, AHEMS can reduce the complexity and resource overhead of the main processor. AHEMS only needs to place a light-weight *runtime monitor* in the pipeline of the *main processor* to send each memory event (memory load/store/allocation/deallocation) to the *security engine* and propagate the metadata of each pointer along with pointer arithmetics. Furthermore, because the metadata propagation is done in hardware, AHEMS has better binary compatibility than any software-only approach (See Chapter 7). Unlike quintupling the size of each register as done by Watchdog (which may lengthen the critical path of the main processor), AHEMS only needs to add $\log N$ bits for each register, where N is the total number of registers. Finally, in contrast to all other approaches, AHEMS stores the metadata in a dedicated physical memory that is inaccessible by the *main processor*. This ensures that the metadata cannot be tampered with by the attacker.

While Patil and Fischer [88] makes the first attempt to offload the checking operations using software to another processor, their approach still incurs a significant runtime overhead (5x slowdown). SafeProc [97] is a hardware approach that allows a checking operation to be delayed. But SafeProc cannot delay a checking operation if there is a register dependence between instructions (See Chapter 8 for more details). *To the best of our knowledge, AHEMS is the first hardware approach that allows completely asynchronous runtime checking to ensure both spatial and temporal memory safety with very low overhead.* Prior study [102] on memory corruption attacks reports that protection schemes that have more than roughly 10% runtime overhead do not tend to get wide deployment in production systems. AHEMS has a better chance to get deployed than other approaches because it incurs runtime overhead as low as 10.6%. Although the asynchronous property of AHEMS may result in delayed detection of an attack, we show that the detection latency is configurable by the size of the FIFO and is generally, if not always, too short for an attacker to launch an attack (see Section 7.2.1).

1.1 Contributions

In summary, this thesis makes the following contributions:

- We propose AHEMS, an architectural support using a pointer-based approach that utilizes *asynchronous checking* to enforce both spatial and temporal memory safety with low runtime overhead. AHEMS incurs an average of 10.6% overhead on Olden benchmarks, which is an order of magnitude lower than four other compared approaches. In addition, AHEMS has good support for binary compatibility, source compatibility, modularity, and scalability to facilitate the progressive deployment of AHEMS on existing applications.

- We propose three options – *in-processor*, *co-processor*, and *external-device designs* – to deploy the *security engine* of AHEMS. This provides system designers with the flexibility on the placement of the *security engine* depending on the system requirements. We compare the pros and cons of these three designs choices.

- We protect the metadata by storing them into different physical memory from that used by the *main processor*. This provides a stronger guarantee on the integrity of the metadata than most existing approaches, which store metadata in the same memory as that used by the main processor.

- We implement a prototype of AHEMS on an FPGA board to evaluate detection coverage, runtime overhead, hardware overhead, power consumption, and impact on the critical path of the main processor. Unlike most other hardware approaches (which use only instruction-accurate or cycle-accurate simulators), the FPGA implementation enables us to provide more accurate (as compared with simulation) evaluation and characterization of the proposed approach. Our experiments show that AHEMS has a negligible impact on the critical path (0.06% overhead) and power consumption (0.5% overhead).

1.2 Thesis Organization

This thesis is organized as follows: Section 2 introduces a general attack model to describe different kinds of attacks in computing systems. Section 3 further focuses on memory corruption attacks and their countermeasures. Section 4 describes the design of AHEMS and how it achieves asynchronous memory safety checking. Section 5 gives a more detailed description of the implementation of AHEMS and addresses some implementation issues. Section 6 shows the experimental results on the detection coverage and additional overheads of AHEMS. Section 7 analyzes the strength and the limitations of AHEMS. Section 8 compares our approach with all other related approaches. Section 9 concludes this thesis and provides directions for the future work.

CHAPTER 2

ATTACKS

In this chapter, a general attack model is defined to formalize the description of security attacks. To show the generality of the model, different types of attacks are given a formal (semi-formal) definition under this model.

2.1 A General Attack Model

A general attack model is one that can describe most attacks. Before defining the attack model, we need to first define the *normal behavior* of a computing system. In our model, there are two major components: *users* and *systems*. A user can interact with a system through the interface it provides. A system services a user's request depending on the current privilege level of the user. If a user wants to perform an operation that is not allowed under his/her privilege, the computing system should dismiss the request. Specifically, a user's activity is a 4-tuple defined as follows:

$$Activity = (Sender, Receiver, Privilege, Action) \tag{2.1}$$

where the *Sender* and the *Receiver* can be either the state of a user or the state of an application in a system. The *Action* is expressed by a parameterized function. This *Activity* means that the *Sender* with the privilege level *Privilege* sends a request to perform the *Action* on the *Receiver*. For instance,

$$(Alice, (Linux1, \texttt{shell}), root, \texttt{runCmd(ls)}) \tag{2.2}$$

means that the user Alice wants to execute an `ls` command on `shell` under the root privilege on the Linux1 machine. A *context* is a list of activities that are performed in chronological order (e.g., $Activity_1$ happens before

Activity$_2$):

$$Context = [Activity_1, Activity_2, ..., Activity_n] \tag{2.3}$$

For example, if Alice wants to log into the Linux1 machine through `ssh` and then performs an `ls` command, then the context of her activities will look like the listing 2.1.

Listing 2.1: An SSH-Login Example of a Context

```
$Context = [Activity_i \text{ where } 1 \leq i \leq 6]$
$Activity_1$ = (Alice, (Linux1, ssh), none, requestForLogin())
$Activity_2$ = ((Linux1, ssh), Alice, none, promptForIDPasswd())
$Activity_3$ = (Alice, (Linux1, ssh), none, login(Alice, 123456))
$Activity_4$ = ((Linux1, ssh), Alice, none, setAsAuthorized(Alice))
$Activity_5$ = (Alice, (Linux1, shell), root, runCmd(ls))
$Activity_6$ = ((Linux1, ls), Alice, root, returnResult(file_list))
```

A context is *executable* if it can happen under the implementation of the system. The *Implementation* for the system S can be defined as the set of all executable contexts in S:

$$Impl(S) = \{C : C \text{ is an executable context in the system } S\} \tag{2.4}$$

A context is *legal* if it obeys all the security principles: Confidentiality, Integrity, Availability, Authenticity, and Non-repudiation. For example, if Alice wants to execute *Activity*$_5$ in the listing 2.1 directly without authenticating herself (by *Activity*$_1$ – *Activity*$_4$), the system should return an error message instead of executing the `ls` command because it violates the authenticity principle. The *Security Specification* (or *Specification* in short) for a system S can therefore be defined as the set of all legal contexts in S:

$$Spec(S) = \{C : C \text{ is a legal context in the system } S\} \tag{2.5}$$

With the above definitions, the relationships among *normal behavior*, *attacks/bugs*, and *unimplemented features* can be clearly illustrated by the Venn Diagram of Implementation and Specification in Figure 2.1. If a context happening among a system and its users is legal and executable, it is called a *normal behavior*. A *general attack* can be defined as a context C in a system

S that is executable but not legal (i.e., $C \in \mathit{Impl}(S)$ and $C \notin \mathit{Spec}(S)$). This definition of general attacks is useful because it can be used to define many kinds of attacks.

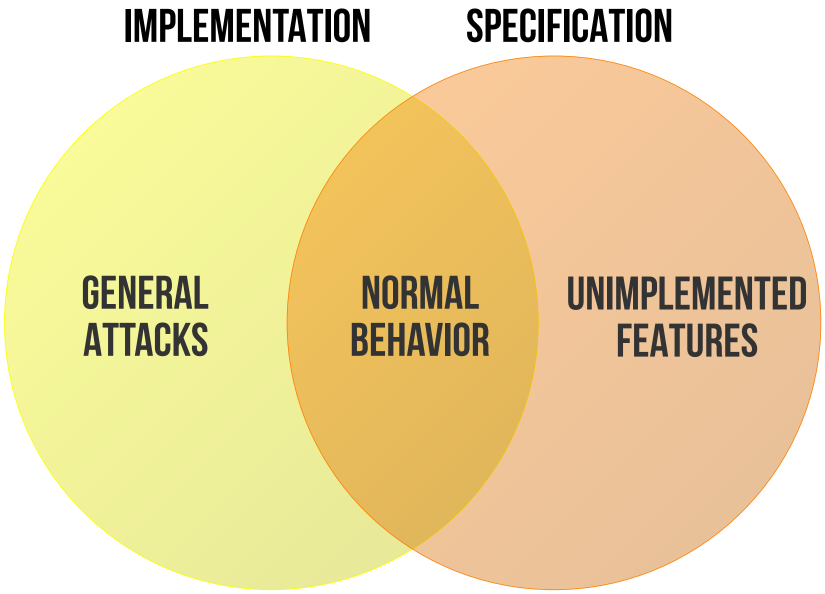


Figure 2.1: The Venn Diagram of Implementation and Specification

2.2 Attack Surface

The attack surface of a computing system is the interfaces it provides to its users. A system can be secured by eliminating its interfaces to reduce the attack surface. However, once a vulnerability is found in the interfaces, an attacker can penetrate the system, since the security of a system relies on its weakest point.

The attack surface can be used to distinguish the impact coverage of an attack. Some attacks may target the network layer, while some may target the application layer. From external to internal, the attack surface can be classified into four layers: *Network*, *Host*, *Application*, and *Data* (as shown in Figure 2.2). Table 2.1 lists different types of attacks according to their

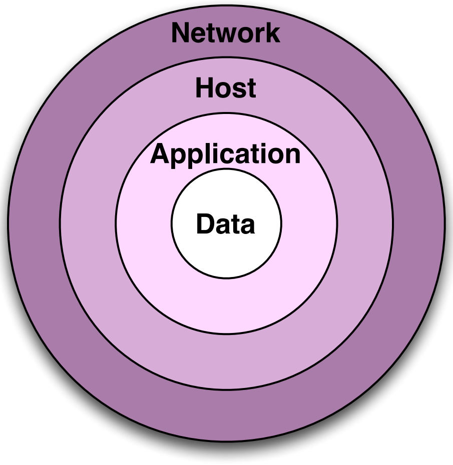


Figure 2.2: Attack Surface in Depth

representative attack surface. Note that an attack may target more than one attack surface, so Table 2.1 categorizes an attack using its most obvious attack surface. Table 2.1 is not an exhaustive list but an enumeration to show representative attacks in each of the attack surface. In next four sections, each type of attack in Table 2.1 is given a formal definition under the attack model in section 2.1 to show the generality of the model.

2.3 Network

In this section, three attacks targeting the network layer are modeled using our general attack model.

2.3.1 DoS (Denial of Service) Attacks

A DoS attack is an attack that tries to disrupt or interrupt the availability of a service or resource to its target users. Since the definition of availability may vary from system to system, the DoS attacks cannot be defined without defining the availability of a system. Given the definition of the availability

Table 2.1: Malicious Attacks Categorized by Attack Surface

Attack Surface	**Attack Type**	**Example Attack**
Network	DoS attacks [a]	DNS amplification attack
	MITM attacks [b]	SSL strip attack
	Protocol vulnerability attacks	DNS spoofing attacks Reflection attacks
Host	Library hijacking attacks [103]	LD_PRELOAD attacks [104]
	Privilege escalation attacks	`/proc/pid/mem` vulnerability [105]
	Hardware vulnerability attacks	DMA attacks [106] Intel `sysret` attack [107]
	Race condition attacks	TOCTTOU attacks [c] [108]
Application	Memory corruption attacks	Control-data attacks Non-control-data attacks [109] Code injection attacks Information leakage attack Integer overflow attacks Format string attacks Buffer overflow attacks Heap spraying attacks Return-to-libc attacks ROP attacks [d] [110] Double-free attacks [111] Use-after-free attacks [112] Zero allocation attacks [113]
	API abuse attacks	Java `ClassFinder` vulnerability [114]
Data	String-oriented attacks	Format string attacks [115] XSS attacks [e] CSRF attacks [f] SQL injection attacks
	Randomness attacks	Low-entropy attacks [116]
	Side channel attacks	Padding oracle attacks [117] Process footprint attacks [118]

[a] Denial of Service attacks

[b] Man-in-the-Middle attacks

[c] Time-of-Check-to-Time-of-Use attacks

[d] Return-oriented Programming attacks

[e] Cross-Site Scripting attacks

[f] Cross-Site Request Forgery attacks

for a specific system S, the DoS attack can be defined as follows:

$$\begin{aligned}
\mathit{Availability}(S) &= \{C \in \mathit{Impl}(S) \mid C \text{ is an available context}\} \\
\mathit{DoSAttack}(S) &= \{C \in \mathit{Impl}(S) \mid C \notin \mathit{Availability}(S)\}
\end{aligned}$$

Note that C is not just a context between a user and a system. It includes all interactions between a system and all its users. To give a specific example, consider a system S with the following criteria of availability:

$$\mathit{Availability}(S) = \{C \in \mathit{Impl}(S) \mid \frac{|\mathit{ServedUser}(\mathit{ActiveUser}(C), C)|}{|\mathit{ActiveUser}(C)|} > 0.95\}$$

where

$$\begin{aligned}
\mathit{ActiveUser}(C) &= \{\mathit{User} \mid \exists\, (\mathit{Sender}, \mathit{Receiver}, \mathit{Privilege}, \mathit{Action}) \in C \\
&\qquad \text{s.t. } \mathit{User} = \mathit{Sender} \text{ or } \mathit{User} = \mathit{Receiver}\} \\
\mathit{ServedUser}(U, C) &= \{\mathit{User} \in U \mid \text{The requests of } \mathit{User} \text{ are all served in } C\}
\end{aligned}$$

That is, C is an available context if requests from 95% of the active users in the context are served.

2.3.2 MITM (Man-In-The-Middle) Attacks

A MITM attack is one way of eavesdropping or tampering with messages between two endpoints by making each endpoint believe that the attacker is the other endpoint it is communicating with. Here the attacker becomes a hidden intermediary who can eavesdrop or manipulate messages between the two endpoints. Since MITM attacks constitute a class of attacks, a subclass of MITM attacks is modeled to provide a more specific example of the general attack model.

Consider a system S with the specification that no user should be in the middle of two communicating users. Let Alice and Bob be the two endpoints that are communicating with each other in the system S, and let Mallory be the attacker. Given a context $C \in \mathit{Impl}(S)$ in the system S that involves the activities of all participants, consider the sub-context[1] C' that includes

[1] Note that a context C is originally a sequence of activities, not a set. But here the order of activities is not important, so we treat it as a set.

only all activities involving Alice and Bob, as follows:

$$\begin{aligned} C' = \{(&Sender, Receiver, Privilege, Action) \in C \mid \\ &(Sender = \text{Alice or Bob}) \text{ or } (Receiver = \text{Alice or Bob})\} \end{aligned}$$

If there exists an activity $(Sender, Receiver, Privilege, Action) \in C'$ such that *Sender* or *Receiver* is Mallory, it is called an MITM attack. Although the above definition of MITM attack is specific to the system S, it shows the effectiveness of our general model to model this kind of attacks.

2.3.3 Protocol Vulnerability Attacks

A protocol vulnerability attack is an attempt to penetrate a system by exploiting the flaws of a protocol. For example, the reflection attack is a kind of protocol vulnerability attacks. This attack utilizes the loophole of a challenge-response authentication system to authenticate the attacker by tricking a legitimate user into providing the response to a challenge. These attacks usually happen when there is a gap between the definition of a protocol (denoted by $ProtoDef(S)$) and its security specification (denoted by $Spec(S)$). S here denotes the system that implements the protocol. As mentioned in Section 2.1, $Spec(S)$ should take into account all security principles. Therefore, a secure protocol should have the following property:

$$(\forall C)(C \in ProtoDef(S) \Rightarrow C \in Spec(S))$$

This means every possible context that can happen in the protocol definition of S is secure. A correct implementation of a protocol should have the following property:

$$(\forall C)(C \in Impl(S) \Rightarrow C \in ProtoDef(S))$$

A protocol vulnerability attack can accordingly be defined as follows:

$$ProtoVulnAttacks(S) = \{C \in Impl(S) \,|\, C \in ProtoDef(S) \wedge C \notin Spec(S)\}$$

Formal methods are utilized to find vulnerability in network protocols by defining the protocol in formal language and exploring contexts that may violate the security specification.

2.4 Host

In this section, four attacks aimed at the host level (e.g., the OS or shared libraries) are modeled using our general attack model.

2.4.1 Library Hijacking Attacks

A library hijacking attack (DLL hijacking attack in Windows environments) is an attempt to override the original dynamically linked library with a malicious library so that the malicious code gets executed when a function in the library is called. A library hijacking attack causes a function to deviate in its behavior from the original definition.

In general, a function f in the system S is a mapping from a system state S_1 to a system state S_2. In the security specification of the system S, each function f in the system S has its *range*[2] for each input state S_1. Let $\mathit{Range}(S, f, S_1)$ be the set of secure output states (under the security specification) for a function f in the system S given the input state S_1. That is, $f(S_1) \in \mathit{Range}(S, f, S_1)$ if f is the original function that follows the specification. A library hijacking attack is said to happen in a context C of a system S if the following property becomes true:

$\exists$ a consecutive subsequence $[(\mathit{Sender}, (\mathit{Receiver}, S_1), \mathit{Privilege}, f),$
$(\mathit{Sender}, (\mathit{Receiver}, f(S_1)), \mathit{Privilege}, g)]$ in C
such that

$$f(S_1) \notin \mathit{Range}(S, f, S_1)$$

This means that there exists a function call f in the context C that changes the system state from S_1 to S_2, where the S_2 is not a secure state according to the definition of f. Note that the receiver in the above property is represented by not only its name but its state.

[2] The *range* of a state is the set of all its possible output states.

2.4.2 Privilege Escalation Attack

A privilege escalation attack is an attempt by an attacker to gain elevated access by exploiting a software or design vulnerability in the system. It enables the attacker to perform unauthorized operations. This attack happens when there exists an action that is authorized by the implementation of the system but not authorized by the security specification of the system. Specifically, let S be a system, $User$ be an user, and $State$ be a system state. The set of actions that $User$ is authorized to perform under the specification or the implementation of S, respectively, can be defined as follows:

$$\begin{aligned}
&\mathit{AuthorizedActions}(User, \mathit{State}, \mathit{Spec}(S)) = \\
&\quad \{PA = (\mathit{Privilege}, \mathit{Action}) \in \mathit{Actions}(S) \mid \\
&\quad User \text{ is authorized to perform } PA \text{ under the specification of } S\} \\
&\mathit{AuthorizedActions}(User, \mathit{State}, Impl(S)) = \\
&\quad \{PA = (\mathit{Privilege}, \mathit{Action}) \in \mathit{Actions}(S) \mid \\
&\quad User \text{ is authorized to perform } PA \text{ under the implementation of } S\}
\end{aligned}$$

where the tuple $(\mathit{Privilege}, \mathit{Action})$ represents an action Action that is performed with the privilege $\mathit{Prvilege}$, and $\mathit{Actions}(S)$ is the set of all possible actions in the system S. A privilege escalation attack is said to happen in the context C when the following condition is true:

$\exists$ an activity $(\mathit{Sender}, (\mathit{Receiver}, \mathit{RecvState}), \mathit{Privilege}, \mathit{Action}) \in C$ such that

1. $(\mathit{Privilege}, \mathit{Action})$ is executed in C
2. $(\mathit{Privilege}, \mathit{Action}) \in \mathit{AuthorizedActions}(\mathit{Sender}, \mathit{RecvState}, Impl(S))$
3. $(\mathit{Privilege}, \mathit{Action}) \notin \mathit{AuthorizedActions}(\mathit{Sender}, \mathit{RecvState}, \mathit{Spec}(S))$

The above condition means that there exists an Action in the context C requested by the Sender and executed on the $\mathit{Receiver}$ with the privilege $\mathit{Privilege}$ when the receiver's state is $\mathit{ReceiverState}$, and the action is authorized under the implementation of the system S but not authorized under the security specification of S. Therefore, the attacker performs an unauthorized action.

2.4.3 Hardware Vulnerability Attacks

A hardware vulnerability attack is an attempt to attack the system by triggering the design flaws or bugs in the hardware used by the system. This attack is possible when there exists a semantic gap between the hardware implementation of the system S (denoted by $HardImpl(S)$) and its hardware security specification (denoted by $HardSpec(S)$). Note that $HardImpl(S)$ is slightly different from the $Impl(S)$ defined in Section 2.1 in that only primitive hardware contexts that are needed to compose all contexts in $Impl(S)$ are included in $HardImpl(S)$, while all possible contexts in the implementation of the system S are included in $Impl(S)$. That is, any context $C \in Impl(S)$ can be decomposed into a concatenation of primitive hardware contexts:

$$\begin{aligned}
&\forall C \in Impl(S), \exists\, C_1, C_2, ..., C_n \in HardImpl(S) \\
&\text{such that } C = [C_1 : C_2 : ... : C_n] \\
&\text{where } [A : B] \text{ is a context composed by the concatenation } A \text{ and } B
\end{aligned}$$

For example, if $C \in Impl(S)$ is a context that a user logs into the system via SSH. Each of $C_1, C_2, ..., C_n \in HardImpl(S)$ will become one assembly instruction in the hardware, and C_1 through C_n together perform the same functionality as C does. One obvious corollary is that $HardImpl(S) \subset Impl(S)$ because $Impl(S)$ includes any possible context. The relationship between $HardSpec(S)$ and $Spec(S)$ is similar to that between $HardImpl(S)$ and $Impl(S)$. With the above definition, a hardware-vulnerability attack happens if

$$\begin{aligned}
&\exists\, C = [C_1 : C_2 : ... : C_n] \in Impl(S) \text{ and } C_1, C_2, ..., C_n \in HardImpl(S) \\
&\quad\text{such that} \\
&\quad(\exists C_i \text{ where } 1 \leq i \leq n)(C_i \in HardImpl(S) \text{ and } C_i \notin HardSpec(S))
\end{aligned}$$

This means that one of the primitive hardware contexts that compose the context C is executable but not legal.

2.4.4 Race Condition Attacks

A race condition attack is an attack that utilizes an unexpected behavior of race conditions in a system to achieve the attacker's goal. A race condition in the software system means that the resultant system state of an execution is dependent on the order or timing of other uncontrollable executions. To model the race-condition attacks, we need to define the *PossibleExecPath* of a context. In a system with a concurrent environment (concurrent users or concurrent threads), an execution of a functionality can have different resultant contexts due to the race condition. Listing 2.2 shows an example of one execution having two different resultant contexts:

Listing 2.2: One Execution Having Two Different Resultant Contexts

```
Context 1:
  $A_1$ = (Bob, (Linux1, program1), Bob, read(A))
  $A_2$ = (Alice, (Linux1, program2), Alice, write(B, 10))
  $A_3$ = (Bob, (Linux1, program1), Bob, read(B))
  $A_4$ = (Bob, (Linux1, program1), Bob, write(C, A+B))
Context 2:
  $A_1$ = (Bob, (Linux1, program1), Bob, read(A))
  $A_2$ = (Bob, (Linux1, program1), Bob, read(B))
  $A_3$ = (Bob, (Linux1, program1), Bob, write(C, A+B))
  $A_4$ = (Alice, (Linux1, program2), Alice, write(B, 10))
```

Thus, the $PossibleExecPath(C)$ is defined as the set of all possible resultant contexts having the same execution as the context C. In the example of Listing 2.2, `Context 1` $\in PossibleExecPath($`Context 2`$)$ and `Context 2` $\in PossibleExecPath($`Context 1`$)$. A race-condition attack is said to happen in the context C if

1. $C \in Impl(S) \land C \notin Spec(S)$
2. $\exists\, C' \in PossibleExecPath(C)$ such that $C' \in Impl(S) \cap Spec(S)$

The first condition implies that there is an attack in the context C because C is executable but not legal. The second condition further implies that the attack is caused by the race condition because there exists a possible execution path that does not violate the security specification.

2.5 Application

In this section, two application-level attacks are modeled using our general attack model.

2.5.1 Memory Corruption Attacks

A memory corruption attack is an attempt to overwrite the contents of a memory location by exploiting the weakness of the programming language used by the program such as the memory unsafety [3] of C/C++ (e.g., array out of bound, use-after-free, or dangling pointers). The property of memory safety requires

1. a memory region is allocated before its access
2. no memory access is allowed after a memory region is deallocated
3. each memory region is associated with a (base, bound) and every memory access should happen in the region (base, base + bound)

Given a context C in the system S, we need to extract all memory activities from the context C to examine its memory safety. The memory activities are one of the following activities:

```
M = (Sender, Receiver, Privilege, malloc(obj, base, bound))
R = (Sender, Receiver, Privilege, read(obj, offset))
W = (Sender, Receiver, Privilege, write(obj, offset, data))
F = (Sender, Receiver, Privilege, free(obj))
```

where `malloc(obj, base, bound)` allocates a memory region (base, base + bound) for the object `obj`; `read(obj, offset)` reads the data from `obj` with the offset `offset`; `write(obj, offset, data)` writes `data` into the `obj` with an offset `offset`; `free(obj)` deallocates the memory region allocated to `obj`.

[3] See Chapter 3 for the definition of memory safety.

With the above definition, a context C is memory safe if

$\forall R = (Send_1, Recv_1, Priv_1, read(obj_1, offset_1)) \in C$

1. $\exists M_1 = (Sender_1, Recv_1, Priv_1, malloc(obj_1, base_1, bound_1)) \in (C$ before $R)$

2. $\nexists D_1 = (Send_1, Recv_1, Priv_1, free(obj_1)) \in (C$ between M_1 and $R)$

3. $0 \leq offset_1 \leq bound_1$

and

$\forall W = (Send_2, Recv_2, Priv_2, write(obj_2, offset_2, data_2)) \in C$

1. $\exists M_2 = (Send_2, Recv_2, Priv_2, malloc(obj_2, base_2, bound_2)) \in (C$ before $R)$

2. $\nexists D_2 = (Send_2, Recv_2, Priv_2, free(obj_2)) \in (C$ between M_2 and $R)$

3. $0 \leq offset_2 \leq bound_2$

where (C before R) is the set of all *Activities* in C that happen before the *Activity* R; similarly, (C between M_1 and R) is the set of all *Activities* in C that happen between M_1 and R.

The above definition of memory safety means that every memory access to a particular object should reside within its base address and boundary, and the object being accessed should be allocated and not deallocated at the time of access. A memory corruption attack is said to happen in the context C if $C \in Impl(S)$ and C is not memory safe.

2.5.2 API Abuse Attacks

Some APIs in the system are not well designed and can be used to attack the system by manipulating the arguments provided to the APIs. This is called an API-abusing attack. One example is the Java `ClassFinder` vulnerability [114], in which several APIs are abused to enable the attacker to execute arbitrary Java code. Let $APIs(S)$ be the set of all API functions in the system S and $Params(func)$ be the set of possible parameter lists for the function $func$. A system S is vulnerable to API-abusing attacks if

$$\begin{aligned}
&\exists \text{ a context } C = [A_1, A_2, ..., A_n] \in \mathit{Impl}(S) \text{ where}\\
&\quad A_i = (\mathit{Sender}_i, \mathit{Receiver}_i, \mathit{Priv}_i, \mathit{API\_func}_i(P_i)) \text{ where}\\
&\qquad \mathit{API\_func}_i \in \mathit{APIs}(S), \quad P_i \in \mathit{Params}(\mathit{API\_func}_i)\\
&\text{such that } C \notin \mathit{Spec}(S)
\end{aligned}$$

This means that there exists a combination of API calls with crafted arguments that can violate the security specification of the system S (e.g., execute arbitrary code provided by the attacker).

2.6 Data

In this section, we model three attacks that are launched by manipulation of data.

2.6.1 String-oriented Attacks

A string-oriented attack is an attempt to feed the victim system with specially crafted inputs that can evade the sanitization policy of the system and thus subvert the system. For example, XSS and CSRF attacks usually exploit the incompleteness of HTML sanitization to embed scripts on a website, while SQL Injection attacks often manipulate the HTTP parameters to inject SQL code. String-oriented attacks are still a great threat in Web Security since fields are often left unsanitized.

Let $\mathit{Params}(func)$ be the set of all possible parameter lists for the function $func$ under the implementation of S. And let $\mathit{LegalParams}(func, S)$ be the set of all legal parameters lists for the function $func$ under the specification of S. A string-oriented attack is said to happen in the context C if

$$\begin{aligned}
&\exists\ (\mathit{Sender}, \mathit{Receiver}, \mathit{Privilege}, func(P)) \in C \text{ such that}\\
&\qquad P \in \mathit{Params}(func) \wedge P \notin \mathit{LegalParams}(func, , SS)
\end{aligned}$$

This means that there exists a malicious input P to the $func$ that will violate the security specification of S.

2.6.2 Randomness Attacks

A randomness attack is an attempt to find a valid token used in the system (e.g., password or session id) by exploiting its low-entropy property to perform unauthorized actions. For instance, Argyros and Kiayias [116] show that the PHP system is vulnerable to randomness attacks due to the low entropy of its pseudo-random number generator. There are many types of randomness attacks. Here we model a randomness attack that involves only an attacker and a victim system. The activities between the attacker and the victim are abstracted into the following two activities to simplify the model:

$$\begin{aligned} &\mathit{Request} = (\mathit{Attacker}, \mathit{Vicim}, \mathit{Privilege}, \mathit{send}(\mathit{request\_msgs})) \\ &\mathit{Reply} = (\mathit{Victim}, \mathit{Attacker}, \mathit{Privilege}, \mathit{reply}(\mathit{response\_msgs})) \end{aligned}$$

A *token* in the system S is vulnerable to randomness attacks if there exists a context $C = [A_1, A_2, ..., A_n] \in \mathit{Impl}(S)$ between the attacker and the victim where A_i is of the form of *Request* or *Reply* such that the information from $\mathit{AllResponses}(C)$ can reduce the entropy of the *token* into *solvable*[4] range where

$$\begin{aligned} \mathit{AllResponses}(C) = \{&\mathit{response\_msgs} \mid \\ &(\mathit{Victim}, \mathit{Attacker}, \mathit{Privilege}, \mathit{reply}(\mathit{response\_msgs}) \in C\} \end{aligned}$$

The above definition of randomness attack means that there exists a context where the response from the victim can reveal enough information for the attacker to guess the token. The information can be revealed either by wrong trials of tokens or any additional message sent back by the victim. For example, the system time that is usually used as a seed for the pseudo-random number generator may be revealed to the attacker.

2.6.3 Side Channel Attacks

A side channel attack is an attack that utilizes the information gained from the indirect sources of the system, to acquire private information or perform unauthorized actions. For instance, the padding oracle attack is a kind of side

[4] *Solvable* here means that the token can be guessed given reasonable trials and time.

channel attack [117] that uses the difference of response time as the indirect information to guess the tokens in the system. Let $Observable(C, User)$ be the set of all activities in the context C that can be observed by the $User$. A side channel attack regarding the $private\_information$ happens in the context $C \in Impl(S)$ if

$$\begin{array}{l} \exists \text{ a user } User \text{ in the system } S \text{ such that} \\ Observable(C, User) \text{ reveals information about the } private\_information \end{array}$$

The above means that a side channel attack happens if combining all the activities observable to the $User$ makes possible for the $User$ to recover the $private\_information$ that should not be known to the $User$.

CHAPTER 3

MEMORY CORRUPTION ATTACKS

This chapter introduces the concepts of *memory errors* and *memory safety* and their relations with memory corruption attacks. Following that, we describe different types of memory corruption attacks to help frame the scope of this thesis. Finally, a variety of defense mechanisms are discussed to delineate the challenges faced by current researchers.

3.1 Overview

A *memory corruption attack* is achieved by exploiting the *memory errors* in a program to corrupt the memory contents targeted by an attacker aiming to gain control over the program. A *memory error* occurs when a pointer being dereferenced does not point to its intended object. That is, the pointer may point to an invalid memory location, a wrong object, or a deallocated memory location. Programming languages, such as C/C++, that support manual memory management (memory allocation/deallocation, pointer arithmetics) are vulnerable to memory errors, while other languages that employ automatic memory management (e.g., garbage collection), such as Java, are memory-safe. A program free of memory errors is *memory-safe*. *Memory safety* and *memory errors* are complementary concepts. Memory safety of a program can be further divided into *spatial memory safety* and *temporal memory safety*. Spatial memory safety requires each dereferenced pointer in a program be initialized and stay within its base address and boundary. Temporal memory safety requires each dereferenced pointer in a program point to an object that still exists (i.e., allocated but not deallocated yet) at the time of dereference. This thesis focuses on enforcing the full memory safety of a program to prevent memory corruption attacks. In the following two sections, we introduce different types of memory corruption attacks that violate either spatial or temporal memory safety.

3.2 Attacks Categorized by Targets

In general, a memory corruption attack can be classified into one of the following categories depending on their attack targets: *control-data attacks*, *non-control-data attacks*, *code injection attacks*, and *information leakage attacks*. The avenue through which an attack overwrites the targets is irrelevant in this categorization.

3.2.1 Control-data Attacks

A control-data attack aims to hijack the control flow of the program by overwriting its control data, including return addresses, function pointers, virtual function tables, GOT (Global Offset Table), PLT (Procedure Linkage Table), and exception-handler pointers. Figure 3.1 is an example program that is vulnerable to control-data attack. The `strcpy(temp, str1)` may overflow the buffer `temp` and therefore overwrite the return address in the stack with the attacker-controlled value. The return address is the most common target among all memory corruption attacks since the return address in the stack of x86 is adjacent to the local buffer and can be easily overwritten if the buffer is overflowed.

```c
void copy_str(char *str1, char *str2) {
  char temp[100];
  strcpy(temp, str1);
  temp[99] = ’\0’;
  strcpy(str2, temp);
}
```

Figure 3.1: An Example of a Control-data Attack

3.2.2 Non-control-data Attacks

A non-control-data attack [109] attempts to control the program execution flow by overwriting the decision-making variables, program configuration, user identity, or input data. The difference between a non-control-data attacks and a control-data attacks is that the former does not violate the control flow integrity because it does not introduce a new execution path, while the latter may not follow the original control flow of the program. Therefore, it is usually harder to detect non-control-data attacks, since control flow integrity does not help. Figure 3.2 is an example of a non-control-data attack. In Figure 3.2, the local decision-making variable `authenticated` may be overwritten by the instruction `strcpy(user,hash(username))`, since `hash(username)` may return a string with length larger than 100. Therefore, by overwriting `authenticated` with value 1, the attacker can login without a correct password.

```c
int auth(char *username, char *password) {
  int authenticated = 0;
  char *user[100];
  authenticated = strcmp(password, get_pw(username));
  strcpy(user, hash(username));
  return authenticated;
}
```

Figure 3.2: An Example of a Non-control-data Attack

3.2.3 Code Injection Attacks

A code injection attack tries to overwrite the code section of a program so that the injected code will be executed when the program counter points to that section. However, most modern operating systems prevent this kind of attack by enforcing code integrity that allows any page to be either writable or executable but not both. Therefore, the executable code section is not writable by the attacker. Unfortunately, self-modifying code or JIT-compiled code (Just-In-Time-compiled code) is still vulnerable to code injection attacks since there is still a short time window in which the code is writable.

3.2.4 Information Leakage Attacks

Instead of corrupting critical data, an information leakage attack [28] attempts to acquire system data or debugging information from the program, such as the address of the `system()` function or the environment variable PATH, so to find vulnerability in the program. The information gained from this attack is usually utilized in another attack to gain control over the program or the system. This attack becomes an important step when ASLR (Address Space Layout Randomization) [20, 23] is deployed in the system. This is because ASLR randomizes the stack, the heap, and the address of each function such that the attacker may need some information leaked from the program to position the attack target or reuse the code in the program. Figure 3.3 is an example of code that is vulnerable to information leakage attacks because it may leak too much information about how the program fails when there is an exception. The attacker can launch an attack that targets the weakness of the program leaked by the stack trace of the exception. For example, Figure 3.4 is an actual MySQL error message from the example in CWE-201 [119].

```
try {
  do_sth();
} catch(Exception ex) {
  ex.printStackTrace();
}
```

Figure 3.3: An Example of an Information Leakage Attack

```
Warning: mysql_pconnect():
Access denied for user:'root@localhost'(Using password: N1nj4)
in /usr/local/www/wi-data/includes/database.inc on line 4
```

Figure 3.4: An Actual MySQL Error Message That Leaks Credentials

3.3 Attacks Categorized by Vulnerabilities

In this section, memory corruption attacks are categorized by the vulnerability they exploit.

3.3.1 Buffer Overflow Attacks

A buffer overflow attack attempts to overrun the boundary of a buffer due to insufficient bounds checking. The attacker writes an amount of data larger than the buffer size, aiming to overwrite the memory region adjacent to that buffer, such as return addresses, function pointers, or decision-making data. If the target buffer is on the stack of a program, it is called stack-based buffer overflow. If the target buffer is in the heap of a program, it is called heap-based buffer overflow. A buffer overflow attack violates spatial memory safety because the pointer to the buffer goes out of bounds. Many mechanisms are proposed to mitigate buffer-overflow attacks. We discuss them in Section 3.4. The examples in Section 3.2.1 and 3.2.2 are both stack-based buffer overflow attacks.

3.3.2 Format String Attacks

Format string attacks occur when an unchecked user input is evaluated as the format parameter of the `printf()` family functions so that the attacker can use `%n` to corrupt any memory location or use `%x` to acquire some system information. It was first discovered by Miller, Fredriksen, So [120] during their fuzz testing for UNIX utility in 1990. In contrast to buffer overflow attacks, which are sequential access and only allow the attacker to overwrite sequential memory regions, format string attacks allow the attacker to overwrite an arbitrary memory location without corrupting other memory locations. This attack also violates spatial memory safety. Figure 3.5 is an example of code that is vulnerable to format string attacks because the `argv[1]` can contain control sequences such as `%n` or `%x` to corrupt memory locations or read system information:

```c
int main(int argc, char *argv[]) {
  printf(argv[1]);
  return 0;
}
```

Figure 3.5: An Example of a Format String Attack

3.3.3 Heap Spraying Attacks

Heap spraying is a technique that injects code or data in the heap so that arbitrary code execution becomes possible. It is usually achieved by a combination of memory deallocation and allocation operations to clean up the heap and spray the attacker-controlled code/data all over the heap so that the chance that the sprayed data/code gets used is higher. A heap spraying attack needs another vulnerability to trigger the use of the code/data sprayed on the heap. Heap Spraying attacks are mostly seen in the exploits of web browsers, since an attacker can use JavaScript to spray the heap and trigger the vulnerability in the browser to use the sprayed data/code.

3.3.4 Return-to-libc Attacks

A return-to-libc attack, which is a type of code reuse attack, is a technique to defeat the protection of non-executable stack[1] by overwriting the return address with the address of one of the library functions that exists in the program (e.g., the `system()` function) and then overwriting the parameters passed to that function to achieve the attacker's goal. The attacker must know the addresses of needed library functions to launch this kind of attack. This becomes very difficult when ASLR is enabled in a 64-bit system since the addresses of library functions are randomized [24]. In 2005, instead of reusing a whole library function, Krahmer [121] starts to reuse a short code snippet to launch an attack, which is the first one that touches the idea of ROP (Return-Oriented Programming) attacks. A return-to-libc attack is a special case of a ROP attack as discussed in the next section.

[1] Non-executable stack prohibits the attacker from executing the shellcode injected on the stack.

3.3.5 ROP Attacks

ROP (Return-Oriented Programming) attacks, proposed by Shacham [110] in 2007, are inspired by the idea of return-to-libc attacks that reuse the code existing in a program. A ROP attack is a technique that enables arbitrary code execution even under the protection of a non-executable stack or code signing. It manipulates the call stack in a program to call small gadgets (carefully-chosen machine instruction sequences) that are part of the library code. Although each gadget performs a simple operation, such as addition, all gadgets together can form a program of Turing-complete functionality [122] [123]. There are even compilers that help translate programs into gadgets [124]. A ROP attack is a kind of control-data attack, so although it can evade the protection of a non-executable stack, it can be alleviated if control-flow integrity [125] is enforced.

3.3.6 Integer Overflow Attacks

An integer overflow occurs when the result of an arithmetic operation goes out of the available range that can be materialized in the storage space. For example, the addition of the two 32-bit integers 2147483647 and 1 becomes -2147483648 because it wraps around when it overflows. An integer overflow attack attempts to bypass sanity checks for integers that are used as offsets or sizes of memory access or memory allocation (e.g., the length argument of `strncpy()`) by triggering the overflow of integers. Integer overflow attacks usually cause violations of spatial memory safety. Figure 3.6 is a snippet of code from the Linux AGP Driver that is vulnerable to integer overflow attacks as reported in CVE-2011-1745 [126]. If Line 7 of Figure 3.6 does not exist, `pg_start` from the user space (controlled by the user) may be so large as to make `pg_start + mem->page_count` overflow. This may cause the sanity check to be passed and the subsequent uses of `pg_start` as the memory offset to violate spatial memory safety.

3.3.7 Double-free Attacks

A double-free attack occurs when `free()` is called twice for the same allocated memory location. The first `free()` deallocates the memory, while the

```
 1 int insert_mem(agp_memory *mem, off_t pg_start, int type) {
 2   int num_entries;
 3   size_t i; off_t j;
 4   // ... (omitted)
 5   // line 7 is added to prevent integer overflow attacks
 6   if (((pg_start + mem->page_count) > num_entries) ||
 7       ((pg_start + mem->page_count) < pg_start))
 8         return -EINVAL;
 9   // ... (omitted)
10   for (i = 0, j = pg_start; i < mem->page_count; i++, j++) {
11     // POTENTIAL FLAWS:
12     // The following write may use unsanitized j, which
13     // is initialized as the pg_start from user space
14     writel(
15       bridge->driver->mask_memory(bridge,
16                                   page_to_phys(mem->pages[i]),
17                                   mask_type),
18       bridge->gatt_table+j);
19   }
20 }
```

Figure 3.6: A Snippet of the AGP Linux Driver with Integer Overflow Vulnerability

```
void double_free() {
  char *ptr1, *ptr2;
  ptr1 = malloc(SIZE1);
  ptr2 = ptr1;
  ...
  free(ptr1);
  free(ptr2);
}
```

Figure 3.7: An Example of a Double-free Attack

```c
void use_after_free() {
  int fail = 0;
  char *ptr1 = malloc(SIZE);
  ...
  if (fail) {
    free(ptr1);
  }
  ...
  printf("%s", ptr1);
}
```

Figure 3.8: An Example of a Use-after-free Attack

second one corrupts the data structure of the heap metadata. This allows an attacker to perform arbitrary memory overwrites or buffer overflow by manipulating the heap metadata and subsequent allocations of memory with the same size [127]. A double-free attack violates temporal memory safety, since the second `free()` uses deallocated memory. Figure 3.7 is an example code that is vulnerable to double-free attack.

3.3.8 Use-after-free Attacks

An use-after-free attack occurs when an allocated memory location is used after it is freed. As in the double-free attack, the attacker can control the freed memory (through the use after it is freed) that may be later treated as another object in the program. Hence, the content of the object is controlled by the attacker. This also violates temporal memory safety, since the memory is used after it is deallocated. CVE-2012-4792 [128] is an use-after-free attack in 2012 for Internet Explorer 6 through 8 that allows the attacker arbitrary code execution. Figure 3.8 is an example of code that is vulnerable to use-after-free attacks.

3.3.9 Zero Allocation Attacks

A zero allocation attack [113] attempts to overwrite certain memory location by triggering the program to call `malloc()` with size 0, which is an undefined behavior in C. If a `malloc()` is called with size 0, then the returned

```
 1 struct user *gen_new_obj(int value) {
 2   user_data = read_user_data();
 3   if (user_data.len > MAX_LEN) {
 4     return NULL;
 5   }
 6   size = user_data.len; // size may be 0
 7   new_user_data = (struct user) malloc(size);
 8   new_user_data.val = value; // Buffer overflow may occur
 9   return new_user_data;
10 }
```

Figure 3.9: An Example of a Zero Allocation Attack

pointer may be NULL or a small chunk of memory, depending on the compiler used. However, when this happens, the returned pointer usually does not point to a memory as large as expected by the programmer, which may later lead to buffer overflow. Integer overflow attacks can also be used to cause the malloc() to be called with size 0. Figure 3.9 is an example of code that is vulnerable to zero-allocation attacks: In Figure 3.9, if the user_data.len is 0, it can pass the sanity check in line 3 and thus be used as the malloc() parameter, which causes 0-sized memory allocation. Supposing that the malloc() returns a 4-byte chunk of memory while each struct user object is 20 bytes, a buffer overflow may occur in line 8. This is because the offset of val in struct user is out of the bound of the 4-byte memory, which allows an attacker to corrupt memory using the attacker-controlled value.

3.4 Countermeasures

In this section, we discuss protection mechanisms against a subset of memory corruption attacks. These mechanisms usually prevent a specific avenue that memory corruption attacks go through to reduce the attack surface as much as possible. However, whenever a new protection comes out, a new attack is soon crafted to defeat it. For example, when a non-executable stack is deployed in the system, return-to-libc attacks and ROP attacks are proposed to bypass this protection. The root cause of memory corruption attacks is the violation of memory safety. This thesis attempts to solve this problem by AHEMS, which ensures both spatial and temporal memory safety.

3.4.1 Stack Integrity

The call stack of a program is one of the most common targets for memory corruption attacks. Many approaches are proposed to enforce stack integrity. The non-executable stack mechanism prevents the attacker from executing any code injected on the stack of the program. It was first developed by Solar Designer [129] in 1997 for the Linux kernel. Following that, the non-executable stack evolves into the more general policy that enforces a page to be either writable or executable but not both. Under the protection of this policy, code and data sections need to be separated. This protection is currently supported by hardware, so it incurs very low runtime overhead. Therefore, this protection has been deployed on most of the modern operating systems (PaX [5,6] in Linux, ExecShield [7] in Red Hat, DEP [8] in Windows, and W$\oplus$X [9] in BSD). However, the protection is still vulnerable to some of the code reuse attacks such as return-to-libc or ROP, since these attacks do not rely on an executable stack to succeed.

Another protection mechanism for the stack is the use of canary value. This is a random value inserted at the end of a buffer on the stack to be checked against a copy whenever a function returns in runtime, making sure no buffer is overrun. This mechanism relies on the property of sequential access of buffer overflow to protect the stack. Therefore, it is still susceptible to attacks that allow arbitrary memory writes (e.g., format string attacks, double-free attacks). StackGuard [10] implements the canary value approach. StackShield [11] further improves the protection of return addresses by copying the return address to an unoverflowable location when a function is entered, and then copying the return address back before a function returns. This guarantees that the return address is always correct, even if buffer is overrun. ProPolice (Stack Smashing Protector) [12] is a low-overhead GCC extension built on StackGuard. It improves StackGuard by protecting not only the return address but all the registers saved in function's prologue and by rearranging array variables to make overflow harder. Unfortunately, these protection schemes cannot defend against information leakage attacks that leak the canary value or the memory location of the return address.

3.4.2 Code Integrity

The code integrity can be ensured by enforcing writable-xor-executable pages with the page protection scheme mentioned in Section 3.4.1. The protection scheme is supported by most modern operating systems in software and modern processors in hardware. The hardware support for code integrity enables its wide deployment because it incurs negligible performance overhead. Unfortunately, although the protection scheme can hamper code injection attacks, it does not prevent code reuse attacks or code injection attacks for JIT-compiled code or self-modifying code.

3.4.3 Heap Integrity

There are three main types of attacks that target the heap of a program: (1) buffer overflow attacks that overwrite adjacent objects in a heap, (2) corruption of heap management metadata, and (3) heap spraying attacks. In the following, we discuss different techniques that protect a program from these heap-based attacks.

Non-executable heap is widely deployed in modern operating systems to prevent code injection attacks that execute code on the heap. However, it does not prevent code reuse attacks, such as heap spraying attacks that spray control or non-control data on the heap.

Canary-based heap protections, such as HeapSentry [13] and Cruiser [14], are proposed to prevent buffer overflow attacks on the heap, but they still do not protect the heap from use-after-free or double-free attacks that violate temporal memory safety by corrupting heap metadata.

Heap metadata protections. Younan et al. [15] place heap metadata in a protected space separated from the allocated memory to protect them from being overwritten. Nonetheless, this does not stop the use of one allocated buffer to overwrite the data of the neighboring buffer in the heap.

Heap layout reorganization. Some defense mechanisms reorganize the heap layout to harden heap integrity. HeapShield [16] replaces the common freelist-based heap with the layout that (1) each chunk is a multiple of memory pages (i.e., page-aligned), (2) each chunk consists of objects of the same size, and (3) there is a page directory that maintains heap metadata. In this way, HeapShield can efficiently calculate the remaining size for each pointer that points into the heap. So HeapShield can provide a safe version of those vulnerable `libc` functions, such as `strcpy()` or `gets()`, by instrumenting them with code that checks the remaining size of the buffer. DieHard [17] is a probabilistic protection that, for each object in the heap, allocates M times as large as the maximum required size for that object where M is a random value in runtime. In this way, the objects in the heap are far apart from one another, which reduces the success rate of buffer overflow attacks. If multiple replicas of a program are run and their results are voted, the success rate for an attack is even lower, as different replicas have different M. Archipelago [18] trades the abundant virtual address space of 64-bit systems for heap integrity. All objects are allocated far from each other in virtual address space to prevent buffer overflow attacks, while usage of physical memory remains the same. DieHarder [19] further releases the requirement of large memory and widens the attack coverage of DieHard by introducing three new features: Sparse Page Layout, Address Space Sizing, and Destroy-on-free, where the idea of Sparse Page Layout is similar to that of Archipelago. Unfortunately, although reorganization of heap layout helps to avoid buffer overflow attacks, the attacker can still corrupt any content in a heap by combining information leakage attacks that reveal the target data location and attacks that allow arbitrary memory writes.

3.4.4 Randomization

Randomizing a program's address space is a probabilistic method to defend against memory corruption attacks, especially code reuse attacks, since an attacker cannot easily find the gadgets or attack targets due to randomization. However, if one part of the program is not randomized, an attacker may still undermine the program. In the following, we discuss different approaches that emphasize randomization of different key areas in a program.
Code randomization. ISR (Instruction-Set Randomization) [21] was originally proposed to defeat code injection attacks. Nonetheless, the writable-xor-executable page protection (as mentioned in Section 3.4.1) later makes ISR unnecessary. IPR (In-place Code Randomization) [22] randomizes basic blocks in the code to defend against ROP attacks with minimal overhead.
Address Space Layout Randomization (ASLR). ASLR [20,23] random-

izes the addresses of heap, stack, library, and code sections of a program each time a program is loaded. It is now widely deployed in modern operating systems to restrict an attack's ability to find the correct address of data or code. Although ASLR is effective in 64-bit systems, ASLR is still vulnerable to derandomization attacks [24] in 32-bit systems due to the low entropy of randomization. Furthermore, if a program is not compiled as a PIE (Position Independent Executable), the code section of the program is not randomized. Unfortunately, most Linux programs are not compiled as PIEs to avoid the additional performance overhead [25]. Since most legacy programs are not compiled as PIEs, Binary-stirring [26] extends the coverage of ASLR to randomize the main code section of a legacy program. Binary-stirring rewrites the legacy program's binary and develops a customized program loader that enables randomization of the main code section without the existence of source code.

Data Space Randomization (DSR). Bhatkar and Sekar propose DSR [27] to randomize the representation of data in the memory by encrypting all variables in a program with multiple keys. DSR can defend against both control-data and non-control-data attacks. Also, DSR has high enough entropy even in the 32-bit systems, as opposed to ASLR. However, DSR requires inter-procedural pointer analysis, which makes it unable to compile different modules of a program separately. Besides, all libraries used by a DSR-protected program need to be recompiled for DSR to work correctly.

Overall, the effectiveness of all randomization approaches rely on the assumption that the content of some data or code cannot be located or recovered by an attacker. However, information leakage attacks may break this assumption and thus bypass these protections [28].

3.4.5 Integer Integrity

There are four main classes of mechanisms that aim to guarantee integer integrity: *static analysis*, *dynamic analysis*, *language support* and *library support*.

Static analysis. Static analysis on the source code of a program can find the integer errors without providing any input to the program. This is because static analysis evaluates every possible execution flow. This has the advantage that even corner cases of integer errors can be found, while the drawbacks are (1) static analysis may report too many false positives (benign integer errors that programmers are aware of or intend to have) and (2) some static analysis approaches are not scalable enough to check integer errors on the Linux kernel. Examples of static analysis approaches are KLEE [29], PREfix + Z3 [30], SmartFuzz [31], IntScope [32], LLBMC [33], and KINT [34].

Dynamic analysis. Dynamic analysis instruments programs with checks to detect the integer errors in runtime. In contrast to static analysis, dynamic analysis approaches produce fewer false positives and thus give more accurate results. However, the detection coverage of integer errors is not 100% and depends on the representativeness of the input fed to the profiling programs that are inserted with checkers. Mechanisms that employ dynamic analysis include blip [35], ARCHERR [36], UQBTng [37], RICH [38], IntFinder [39], IntPatch [40], and IOC [41]. GCC also has an option `-ftrapv` and a `size_overflow` plugin [42] to detect integer overflows.

Language support. Some languages, such as Python, support infinite-precision integers that can eliminate all integer errors. However, infinite-precision integers may not be suitable for all programs because the implementation incurs more performance overhead than fixed-width integers.

Library support. There are also libraries (e.g., SafeInt [130], IntegerLib [131]) that help programmers avoid integer errors if they use the APIs provided by the libraries to perform integer arithmetics. The libraries perform checks for the programmers to detect integer errors and call default handlers when there is an integer error.

3.4.6 Control Flow Integrity (CFI)

Control Flow integrity (CFI) ensures that any execution path of a program in runtime conforms to the control flow graph computed according to its source code semantics. In fact, only indirect control transfers (e.g., indirect calls/jumps/returns) need to be checked for CFI if code integrity is enforced. This is because the target addresses of direct control transfers are fixed in the read-only code segment. CFI can hamper any attack that attempts to divert a program's control flow to an illegal one, especially control-data attacks and ROP attacks. However, depending the accuracy of the CFI, some return-to-libc attacks can evade the protection of CFI. Also, CFI cannot defeat non-control-data attacks, since non-control-data attacks do not take an illegal execution path in runtime. Challenges of CFI are (1) performance overheads, (2) false positives and negatives, (3) source code availability, and (4) binary compatibility. Although numerous techniques are proposed to address these issues, there is no solution that resolves all of them. Examples are Program Shepherding [43], CFI [44, 45], WIT [46], HyperSafe [47], Control-flow locking [48], CPM [49], CFIMon [50], MoCFI [51], FPGate [52], CCFIR [53], and Total-CFI [54]. SFI (Software Fault Isolation) [132–135] enforces data sandboxing that allows control transfers only to targets within the sandbox, which can be viewed as a weak version of CFI.

3.4.7 Data Flow Integrity (DFI)

Similar to but different from CFI, data flow integrity (DFI) [73] enforces that the flow of data in runtime conforms to the data flow in the control flow graph statically computed by reaching definition analysis. For each use of a variable (the statement that loads the variable), only a specific set of the variable's definitions (the statement that stores value to the variable, defs for short) can reach it in the control flow graph. DFI aggregates those defs in the same group and assigns it an ID. Therefore, for each use of a variable, that variable should always be defined by defs in the same group (i.e., have the same ID) when there is no memory error. In this way, DFI only needs to associate the group ID to a variable when a variable is defined. It then checks at each use of a variable to see if the group ID of variable is in the set of the acceptable group IDs specific to that use. This enforcement ensures each use of a variable reads the value written by the correct group of defs, which prevents most memory corruption attacks. However, the imprecision caused by treating different defs as the same group may allow the attacker to craft an attack that escapes detection. If a store instruction at some point of the program is the def for many different variables, those variables share the same group ID and thus the store instruction can be utilized by an attacker to overwrite the variable of the same group that is not intended by the program. For example, the system calls `write()` and `read()` are called very often to read and write files. The store instructions in `write()` are usually allowed to write to many variables, which can be exploited by the attacker. Furthermore, unmodified libraries cannot interoperate with DFI-protected binaries, which makes incremental deployment difficult.

3.4.8 Information Flow Integrity

Information Flow Integrity (IFI) enforces that no direct information from user inputs is propagated to the critical data, such as return addresses or function pointers, by employing Dynamic Information Flow Tracking (DIFT) [55]. For example, to prevent control-data attacks, a user's input should never be dereferenced as an address. DIFT taints every piece of data that comes from user input, since they are untrusted and can be malicious. Then DIFT propagates the taints depending on the operations performed on the tainted data and the policy used by an approach. If the taint reaches critical data that should not be tainted, DIFT raises an alarm. However, if DIFT propagates for every operation, it may cause too many false positives, whereas it may have false negatives if the propagation policy is too restrictive [136–138]. Raksha [58] attempts to address this issue by proposing an architecture support that allows flexible configuration of information flow policies and enables the detection of high-level attacks, such as SQL injections or XSS (Cross-Site Scripting). Nonetheless, Raksha cannot totally eliminate false positives and false negatives when multiple applications with different design assumptions are run concurrently. For example, suppose there are two applications running concurrently. During a table lookup, one of them should propagate taints to avoid false negatives, while the other should not propagate taints to avoid false positives. In this case, Raksha must generate either false positives or false negatives. Furthermore, DIFT incurs prohibitively high performance overhead if done purely in software. Even the fastest dynamic tainter, Minemu [68], incurs a slowdown of 2.4x for SPECINT 2006. Although architecture supports for DIFT (e.g., Minos [56], RIFLE [57], Chen et al. [139], Infoshield [140] and Raksha [58,59]) are proposed to speed up dynamic tainting, no hardware vendor has adopted these approaches yet. Hence, many research efforts are still devoted to improving the performance and the accuracy of

DIFT in software, including TaintCheck [60], LIFT [61], HiStar [62], TaintTrace [63], Dytan [64], Speck [65], Aftersight [66], PTT [67], Minemu [68], DTA++ [69], TaintEraser [70], Taint-exchange [71] and libdft [72],

3.4.9 Memory Safety

Full memory safety, including spatial memory safety and temporal memory safety, requires that no memory error occurs in a program. This can prevent all types of memory corruption attacks. For the past three decades, a plethora of approaches have been proposed to enforce full or partial memory safety. However, they still suffer from either prohibitively high performance overhead, low source compatibility, low binary compatibility, no modularity support[2], or incompleteness of memory safety, so that none of them is widely adopted in real-world applications. AHEMS, proposed in this thesis, attempts to address these issues by guaranteeing both spatial and temporal memory safety for a program with low performance overhead, high source compatibility, high binary compatibility and modularity support. Chapter 8 discusses and compares memory safety approaches in detail.

[2] Modularity support means that independent components can be processed separately, which enables incremental deployment of the approach. For example, modified binaries can be used with unmodified libraries.

CHAPTER 4

APPROACH

In this chapter, we first describe the concepts adopted by AHEMS to enforce memory safety. Secondly, a high-level operation overview of AHEMS is outlined to show how AHEMS works as a whole. Thirdly, the detailed mechanisms of AHEMS are described to concretize the design. Finally, an illustrative example is given to demonstrate the working scenarios and the effectiveness of AHEMS.

4.1 Full Memory Safety Enforcement

Full memory safety for a program requires that

1. a memory region is allocated before it is accessed,

2. no memory access to a memory region is allowed after it is deallocated, and

3. each memory region is associated with a base address and a boundary, denoted by `(base, bound)`, when it is allocated, and every memory access should occur between `(base, base + bound)`.

The first two points depict temporal memory safety, while the third point describes spatial memory safety. In the following, we focus on memory safety in the C language because C is the most widely used programming language that is notoriously memory-unsafe. According to C semantics, each object, if stored in the memory, should have a base address and a boundary, which identify the memory region used by the object. The object can be

1. a primitive type of data (e.g., `5 (int num), 'a' (char ch)`),

2. a pointer (e.g., 0xDEADBEEF (`int*` ptr)),

3. a struct (e.g., {"Toyota", "Corolla"} (struct car)), or
4. an array (e.g., {1, 2, 3, . . . , 100} (`int ary[100]`)).

Each object is accessed by reference, including

1. direct access (`sum = num + 5; ch = 'c';`),
2. pointer dereferences (`sum = *ptr + 10;`),
3. struct field access (e.g., `car.brand = "Honda";`), or
4. array access (e.g., `ary[1] = ary[2] + ary[3];`).

Every reference has its own intended object. If a reference is used to access an unintended object, it violates spatial memory safety, whereas if a reference is used to access an object that is dead (i.e., uninitialized or deallocated) at the time of access, it violates temporal memory safety. Consequently, to enforce full memory safety, AHEMS conceptually does the following three things: (1) associates each reference with the `(base, bound)` of the intended object when the object is live (allocated and not deallocated), (2) associates the reference with an invalid `(base, bound)` when the intended object is dead, and (3) checks in the case of an object dereference whether the associated `(base, bound)` is valid and whether the accessed memory address is within the `(base, base + bound)`; This is to guarantee temporal memory safety and spatial memory safety, respectively.

4.2 Operation Overview

AHEMS has two major parts: (1) the hardware that is responsible for the efficient runtime checking for memory safety, and (2) the source-code instrumentation that establishes the interface between the program and the hardware. Figure 4.1 illustrates the architecture of AHEMS. Here we briefly describe how the hardware and the instrumented code are coordinated for memory safety checking.

Source Code Instrumentation. Before a program is compiled, its source code needs to be instrumented with `alloc` and `dealloc` instructions by the CIL source-to-source compiler [141]. This is to enable the runtime notification of the hardware about the memory allocation and deallocation events.

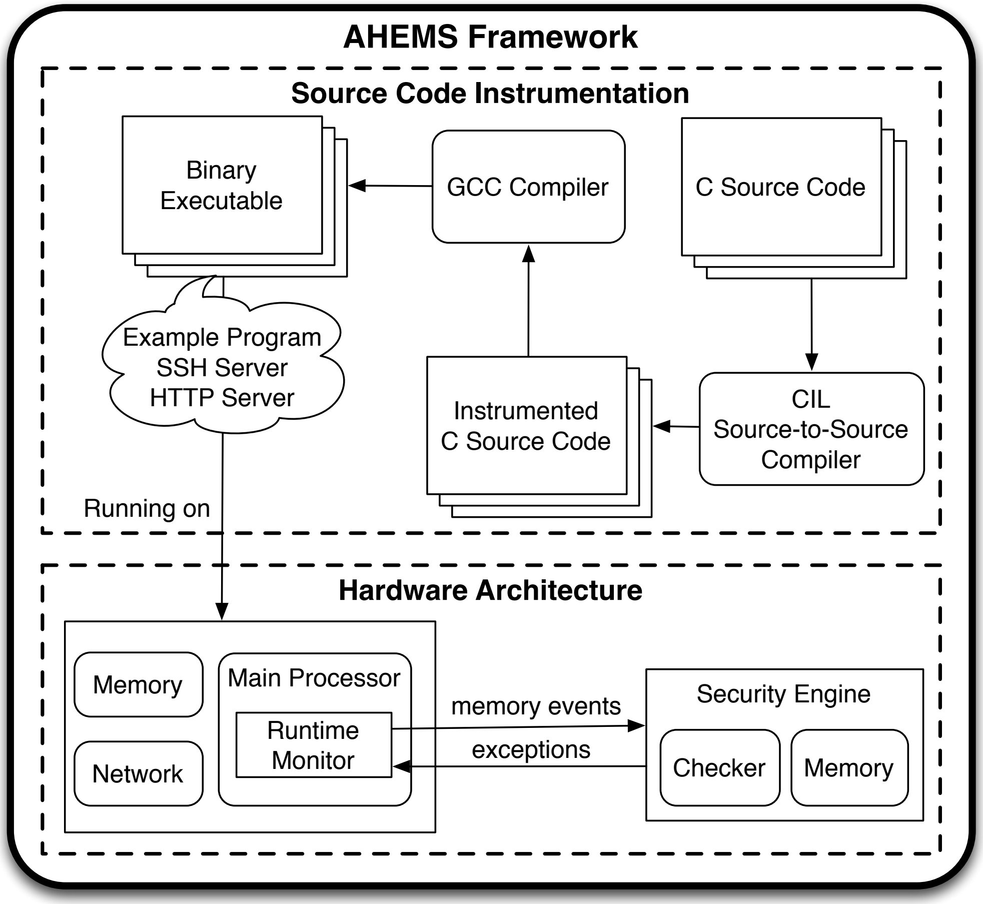


Figure 4.1: AHEMS Architecture

The instrumented code is then compiled with the GCC compiler to generate a binary executable.

Hardware. There are three main hardware components: *main processor*, *runtime monitor*, and *security engine*. The *main processor* is responsible for executing the program. When a program is being executed, the *runtime monitor*, which is embedded in the *main processor*, collects every memory event, including memory load/store/allocation/deallocation and sub-object creation, from the *main processor* and sends that event to the *security engine* for memory safety checking. The *security engine* keeps track of memory events and checks whether each memory operation is legal (i.e., whether an access is out of bound or whether the accessed object exists). If a violation of memory safety is detected, the *security engine* raises an exception, which is communicated to the *main processor* to either stop the program or invoke

```
// return the new pointer associated with (base, bound)
void* alloc(void *base, size_t bound);
// mark the reference as dead
void dealloc(void *reference);
// Create a pointer to an sub-object associated
// with (subbase, subbound)
void* subcreate(void *subbase, size_t subbound);
```

Figure 4.2: Definition of `alloc()` and `dealloc()`

an exception handler. Note that the *main processor* does not need to wait for any information from the *security engine* except for the exceptions, which are rare in normal (i.e., error-free) execution. This enables the *security engine* to perform the checking asynchronously and greatly reduces the runtime overhead.

4.3 Source Code Instrumentation

The goal of source code instrumentation is to inform the hardware about the range and the lifetime of each reference in a C program, including birth (allocation), death (deallocation), and range (base and bound). Therefore, three utility functions `alloc(...)`, `dealloc()`, and `subcreate(...)` are created to notify the hardware about each memory allocation, memory deallocation, and sub-object creation, respectively. (see the definition in Figure 4.2.) `alloc(base, bound)` notifies the hardware to associate the pointer `base` with the range `(base, bound)` and then returns the newly associated pointer. `dealloc(reference)` notifies the hardware that the pointer `reference` is not valid anymore. `subcreate(subbase, subbound)` further shrinks the base and bounds of the sub-object `subbase` in a structure to `(subbase, subbound)`. The CIL compiler [141] is utilized to instrument the source code. CIL is a source-to-source compiler that facilitates static analysis, transformations, and optimizations of C programs. To track the birth and death of every reference, we need to instrument (1) the `malloc()` and `free()` functions for tracking the dynamically allocated references, (2) local variables and function parameters, (3) global and static variables, (4) the sub-objects in a structure, and (5) integer-to-pointer casts. Note that these

```
void *malloc(size_t size) {
  // ... (omitted)
  // compute the return value ret_val
  return alloc(ret_val, size);
}
void free(*ptr) {
  // ... (omitted)
  dealloc(ptr);
}
```

Figure 4.3: Instrumentation of `malloc()` and `free()`

instrumentations do not open potential security holes because they simply wrap around those references with `alloc()`, `dealloc()` and `subcreate()` functions.

4.3.1 `malloc()` and `free()`

AHEMS instruments the `malloc()` and `free()` functions with `alloc()` and `dealloc()`, respectively, as in Figure 4.3. In this way, the hardware is informed of every memory allocation and deallocation event.

4.3.2 Local Variables and Function Parameters

To track local variables and function parameters, AHEMS (1) inserts an additional local pointer `ptr` for each local array, each local variable whose address is taken[1], and each function parameter whose address is taken, (2) inserts `alloc()` in the prologue of the function to initialize `ptr` with base and bound, (3) replaces all uses of the local array with the added local pointer that points to the local array, (4) replaces each use of local variable or function parameter where its address is taken with the added pointer that points to the local variable or function parameter, and (5) inserts `dealloc()` in the epilogue of the function to mark `ptr` as dead. Note that AHEMS does not track local variables or function parameters whose addresses are not taken because their access are always memory-safe. Figure 4.4 shows an example

[1] In C, the address can be taken using the `&` operator.

```
void foo(int in1) {
  int lvar = 5, *ptr, *ptr2;
  int lary[100];




  ptr = &lvar;
  ptr2 = &in1;
  sum = lary[5];
  // use sum and ptr
}
```

```
void foo(int in1) {
  int lvar = 5, *ptr;
  int lary[10];
  int lptr1 = alloc(&lvar, 4);
  int lptr2 = alloc(lary, 40);
  int lptr3 = alloc(&in1, 4);
  ptr = lptr1;
  ptr2 = lptr3;
  sum = lptr2[5];
  // use sum and ptr
}
```

(a) Before Instrumentation

(b) After Instrumentation

Figure 4.4: An Example of Local Variable Instrumentation

program for local variable and function parameter instrumentation.

4.3.3 Global Variables and Static Variables

The tracking of global or static variables is similar to that of local variables except for the insertion positions of the `alloc()` functions and there being no need to insert `dealloc()` functions. Global or static variables are live through the program execution time, so the `alloc()` functions are inserted in the prologue of the `main` function instead of in the function where variables are located. Also, global or static variables do not need to be marked as dead in the epilogue of the `main` function. To give an example, Figure 4.5 illustrates how AHEMS instruments the global variables.

4.3.4 Sub-objects in the Structure

AHEMS instruments a sub-object of a structure with the `subcreate()` function when the sub-object is an array or the address of the sub-object is taken; this associates the sub-object with a sub-base and sub-bound. The lifetime of a sub-object is the same as that of its parent object, so `dealloc()` is not needed for sub-objects. Figure 4.6 shows an example program before and after the sub-object instrumentation. Associating the sub-base and sub-bound for a sub-object can prevent sub-object overflows, which occur when

```c
int gary[100];
int gvar;
int main () {


  int *ptr = &gvar;
  int sum = gary[5];
  // use sum and ptr
  return 0;
}
```

(a) Before Instrumentation

```c
int gary[100], *gptr1;
int gvar, *gptr2;
int main () {
  gptr1 = alloc(gary, 400);
  gptr2 = alloc(&gvar, 4);
  int *ptr = gptr2;
  int sum = gptr1[5];
  // use sum and ptr
  return 0;
}
```

(b) After Instrumentation

Figure 4.5: An Example of Global Variable Instrumentation

one field in a structure overflows another field in the structure. For example, in line 9 of Figure 4.7, `node.id[11]` overflows the internal array of `Node` and overwrites the value of `node.ptr`. However, some common programming idioms may intentionally overflow sub-objects. Figure 4.7 shows an example of intentional sub-object overflows. In line 12 of Figure 4.7, the `memcpy()` may use a pointer of type `char*` to iterate through the entire structure of `node` to copy each byte of `node` to the allocated memory pointed by `node2`, which overflows all sub-objects of `node` and `node2`. Therefore, to eliminate false alarms, AHEMS omits some instrumentation of `subcreate()` at places where intentional sub-object overflows may happen.

4.3.5 Integer-to-pointer Casts

For some low-level C code, such as kernel code or device drivers, integer-to-pointer casts are needed to hardcode some fixed addresses used by the kernel or the device. However, this may cause false alarms in AHEMS because an integer is by default associated with an invalid `(base, bound)`. When an integer is converted into a pointer and dereferenced, the `(base, bound)` of the integer is checked and thus results in a false alarm. To allow some use cases of integer-to-pointer casts, AHEMS allows programmers to manually insert the `alloc()` function to notify AHEMS that this integer is a valid address with a programmer-specified base and bound. Figure 4.8 shows an example of instrumentation that safely casts integers to pointers with `alloc()`.

```
struct Node {
  int nm[10];
  int *ptr;
  int value;
};
void foo() {
  struct Node n;




  n.nm[5] = 3;
}
```

(a) Before Instrumentation

```
struct Node {
  int nm[10];
  int *ptr;
  int value;
};
void foo() {
  struct Node n;
  struct Node *np;
  int (*nmp)[10];
  np = alloc(&n, sizeof(n));
  nmp = subcreate(&np->nm,40);
  (*nmp)[5] = 3;
}
```

(b) After Instrumentation

Figure 4.6: An Example of Sub-object Instrumentation

```
struct Node {
  int num[10];
  int *ptr;
  int value;
};
void foo() {
  struct Node node;
  // an example of sub-object overflows
  node.num[11] = ...;
  // an example of intentional sub-object overflows
  struct Node *node2 = malloc(sizeof(struct Node));
  memcpy(node2, &node, sizeof(node));
}
```

Figure 4.7: An Example of Sub-object Overflow

```
void foo() {
  int *ptr;
  ptr = (int*) 0x12345678;
  ptr[5] = 0xDEADBEEF;
}
```

(a) Before Instrumentation

```
void foo() {
  int *ptr;
  ptr = alloc(0x12345678, 40);
  ptr[5] = 0xDEADBEEF;
}
```

(b) After Instrumentation

Figure 4.8: An Example of Integer-to-Pointer Instrumentation

4.4 Hardware Architecture

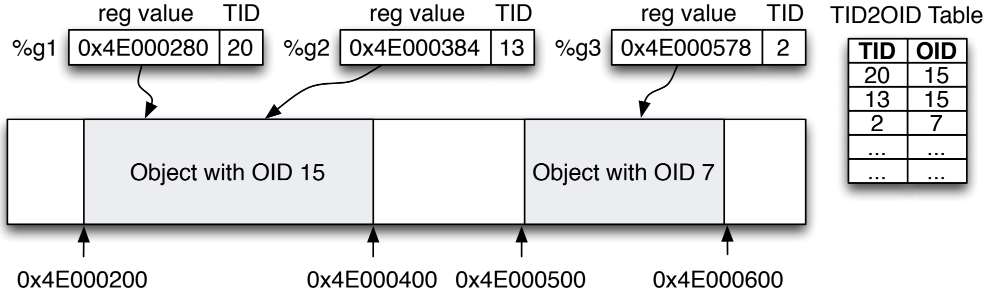
AHEMS instruments the source code of a program so that each memory-unsafe reference in a C program is materialized as a pointer with (base, bound). The goal of the hardware architecture is to ensure that, for each executed load or store instruction, the accessed memory address, which comes from a pointer in the register plus an offset, is within the (base, base + bound). The following subsections describe the hardware design of AHEMS to achieve this goal.

4.4.1 alloc, dealloc, and subcreate Machine Instructions

To implement the alloc(), dealloc(), and subcreate() functions that notify the hardware about the memory events, AHEMS adds the three new machine instructions alloc, dealloc, and subcreate to the main processor. (see next chapter for implementation details.) This enables the runtime monitor to know about every memory allocation, memory deallocation, and sub-object creation event when the alloc, dealloc, subcreate instructions are executed, respectively. (see Table 4.1 for the detail functionality of alloc, dealloc and subcreate instructions.)

4.4.2 Temporary Identifier and Object Identifier

Each register in the main processor is associated with an *temporary identifier* (TID). If a register contains a pointer in the program, its TID is used by the security engine to identify the object to which the register is pointing. Specifically, the security engine uses the TID to look up the object identifier (OID), which is unique for an object, in the security engine's TID2OID lookup table. Figure 4.9 illustrates the relationship between the register values, the TIDs, and the OIDs. If two registers point to different objects, they should have different OIDs in the TID2OID table. Conversely, if two registers point to the same object, they may have different TIDs, but they should have the same OID even if their offsets into the object are different. With the TID for each register and the TID2OID table, the security engine can easily identify the object pointed to by a register when the register is used as an address for a load or store instruction. The security engine can then find the object's



TID	OID
20	15
13	15
2	7
...	...
...	...

Figure 4.9: Relation of Register Values, TIDs and OIDs: register %g1 and %g2 point to different offsets of the same object with OID 15, while register %g3 points to a different object with ID 7. From the TID2OID table, it can be seen that although %g1 and %g2 have different TIDs, they are linked to the same OID.

(base, bound) to check whether the pointer is within legal range to enforce memory safety. Hence, the key point is how to maintain the validity of the TID and the TID2OID table efficiently at any point within a program.

4.4.3 TID and OID Generation and Propagation Rules

For each of the six operations in the main processor (memory allocations, pointer arithmetics, memory deallocations, sub-object creations, loads, and stores), additional actions need to be executed along with the original operation to maintain the relationship between OID and TID and to check for memory safety. Table 4.1 summarizes these actions and their actors. In the following, we describe each of the actions in detail.

Memory allocation. When an object (or a region of memory) is allocated, an alloc instruction is executed to (1) associate the pointer that points to the object with a new TID (Step 1 in Table 4.1), (2) notify the security engine to assign a new OID to that object and map the new TID to the new OID in TID2OID table (Step 2 in Table 4.1), and (3) associate (base, bound, NOPARENT) information of the object with the OID in the OID2BaseBound table, where NOPARENT means that this object is not a sub-object. These actions provide the *first pointer*[2] that points to an object with a new TID. The security engine can use this TID to look up the status of the object by

[2] the pointer obtained at the time of allocating memory for a given object

Table 4.1: TID and OID Generation and Propagation Rules

alloc **Instruction**	**Actor**
ALLOC rd, base, bound	Main Processor
1. rd.tid = tid = tid_gen();	Runtime Monitor
2. TID2OID[tid] = oid = oid_gen(); 3. OID2BaseBound[oid] = (base, bound, NOPARENT);	Security Engine
Pointer Arithmetics	**Actor**
ADD rd, rs1, rs2	Main Processor

if rs1.tid != INVALID then rd.tid = rs1.tid;
else rd.tid = rs2.tid;
	Runtime Monitor

ADD rd, rs1, imm	Main Processor
rd.tid = rs1.tid;	Runtime Monitor
MOV rd, rs	Main Processor
rd.tid = rs1.tid;	Runtime Monitor
dealloc **Instruction**	**Actor**
DEALLOC rd	Main Processor
1. oid = TID2OID[tid]; 2. OID2BaseBound[oid] = INVALID; 3. TID2OID[tid] = INVALID;	Security Engine
subcreate **Instruction**	**Actor**
SUBCREATE rd, subbase, subbound	Main Processor
1. rd.tid = tid = tid_gen();	Runtime Monitor
2. parent_oid = TID2OID[subbase.tid]; 3. TID2OID[tid] = oid = oid_gen(); 4. OID2BaseBound[oid] = (subbase,subbound,parent_oid);	Security Engine
store **Instruction**	**Actor**
STORE [addr], rs	Main Processor
1. addr_oid = TID2OID[addr.tid]; 2. (base,bound,parent_oid) = OID2BaseBound[addr_oid]; 3. if (base,bound,parent_oid) = INVALID then raise exceptions 4. if (parent_oid != NOPARENT and OID2BaseBound[parent_oid] == INVALID) then raise exceptions 5. if (addr < base) or addr > (base + bound) then raise exceptions 6. MEM2OID[addr] = TID2OID[rs.tid];	Security Engine
load **Instruction**	**Actor**
LOAD rd, [addr]	Main Processor
1. rd.tid = tid_gen();	Runtime Monitor
2. addr_oid = TID2OID[addr.tid]; 3. (base,bound,parent_oid) = OID2BaseBound[addr_oid]; 4. if (parent_oid != NOPARENT and OID2BaseBound[parent_oid] == INVALID) then raise exceptions 5. if (base, bound) = INVALID then raise exceptions 6. if (addr < base) or addr > (base + bound) then raise exceptions 7. TID2OID[rd.tid] = MEM2OID[addr];	Security Engine

the TID2OID and OID2BaseBound table when the pointer is dereferenced.

Pointer arithmetics. To make sure each pointer is associated with the correct object, AHEMS needs to propagate the TID when pointer arithmetics is performed. (see Table 4.1.) For example, if the instruction "ADD rd, rs, imm" is performed (i.e., rd = rs + imm) and rs represents a pointer to an object, the destination register rd should also point to the same object as rs. Therefore, the TID of rs should be copied to the TID of rd. Since source code instrumentation enforces that all subsequent pointers that point to an object are derived from the *first pointer* to the object, every subsequent access to the object can always use the propagated TID to look up the OID in TID2OID table for checking the object's liveness and boundary.

Memory deallocation. When an object is freed, the dealloc instruction is executed to invalidate the object pointed by the pointer so that subsequent access to the object triggers an alarm in AHEMS. Specifically, dealloc instruction notifies the security engine to mark TID2OID[tid] and OID2BaseBound[oid] as invalid. If more than one TID is mapped to the same object with OID oid such as the example in Figure 4.9 and the object is deallocated, then all of those TIDs will turn out mapping to an invalidated object. This is because the security engine looks up the entry OID2BaseBound[oid] to see if the object with OID oid is invalidated. Since all of those TIDs map to the same oid, the object pointed by those TIDs is invalidated.

Sub-object creation. Creating an pointer to a sub-object is similar to memory allocation except that the sub-object should also be associated with the OID of its parent object (steps 2 and 4 in Figure 4.1) so that the security engine can use the OID of its parent object (denoted by parent_oid) to determine the lifetime of the sub-object on loads and stores by checking if OID2BaseBound[parent_oid] is INVALID. Therefore, the OID of the sub-object is associated with (base, bound, parent_oid) instead of (base, bound, NOPARENT) in the OID2BaseBound table.

Store. Two additional actions are required along with a store instruction: (1) checking the liveness and boundary of the accessed object and (2) mapping in MEM2OID table the memory address (addr) to the OID of the object pointed by the register rs. For the first action, the information of the accessed object is found by going through the lookup process: addr.tid → oid → (base, bound) (steps 1 and 2 in Table 4.1). Then the validity of (base, bound, parent_oid) is checked to see if the object is

still live (steps 3 and 4 in Table 4.1). Finally, `addr` is compared with the `(base, bound)` to see if it is within the range of the object (step 5 in Table 4.1). If the object is not live or the `addr` is out of bound, the security engine raises an exception. For the second action (step 6 in Table 4.1), the `MEM2OID` table enables the security engine to recover the mapping between TID and OID when the register being stored into the memory is loaded back to the register. (see next paragraph for details.)
Load. A `load` instruction checks the liveness and boundary of the accessed object in the same way the `store` instruction does. In addition, two other actions are performed: (1) associating the destination register (`rd`) with a new TID (step 1 in Table 4.1) and (2) associating the new TID with the OID of the pointer stored at the memory address `addr` (step 7 in Table 4.1). The `load` instruction loads the pointer stored at `addr` into the destination register (`rd`). These two actions recover the association between the stored pointer and the object pointed to by the stored pointer so that the AHEMS can continue to keep track of a pointer even if the pointer is stored into memory and loaded back for later use.

4.4.4 Asynchronous Checking

The reason AHEMS needs both the TID and OID is that they make the asynchronous checking of memory safety possible. With asynchronous checking, the main processor does not need to stall to wait for the result of checking or for the storing/loading of metadata before it can move on to execute the next instruction. The processor only stalls when the queue for checking is full. If each register is associated directly with an OID instead of a TID, the main processor needs to wait for the loading of the OID from the `MEM2OID` table to the destination register `rd` when a `load` instruction is executed. Otherwise the register `rd` is not associated with the correct `OID`. To solve this problem, a new TID `tid` is generated and assigned to `rd.tid` when a `load` instruction is executed. This enables the main processor to continue its execution without worrying about the OID. The `tid` is then asynchronously associated with the OID loaded from the `MEM2OID` table by security engine to help `rd` find the OID of its associated object.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4   int *a = (int *) malloc(sizeof(int) * 15);
5   int *b = a + 20;
6   printf("%08X\n", *b);
7   return 0;
8 }
```

Figure 4.10: An Example `outbound.c` That Violates Memory Safety

4.5 An Illustrative Example

In this section, we will demonstrate how AHEMS tracks the execution of the program `outbound.c` in Figure 4.10 to detect the violation of memory safety. The program `outbound.c` is an example that has an out-of-bounds memory access in line 6 of Figure 4.10. AHEMS should raise an exception when the program reaches line 6. Figure 4.11 is the assembly code of `outbound.c` on SPARC Architecture.

The step-by-step execution is as follows:

1. `40001924:  save %sp, -104, %sp`
   Allocates a stack frame for `main` function.

2. `40001928:  mov 0x3c, %o0`
   Sets `%o0` to `0x3c` to prepare the first argument for the `malloc` function.

3. `4000192c:  call 400019bc <malloc>`
   Calls the `malloc` function with `%o0` as the size and returns the address of the allocated memory in `%o0`. Let's assume the function returns 0x50000000.

4. `40001930:  nop`
   Do nothing.

5. `40001934:  mov %o0, %g1`
   Moves the return value `%o0` to `%g1`. That is, `%g1 = %o1 = 0x50000000`.

6. `40001938:  alloc %g1, 0x3c, %g1`
   (a) `%g1.tid = tid_gen() = #1;`

(b) TID2OID[#1] = oid_gen() = #1;
(c) OID2BaseBound[1] = (%g1, 0x3c, NOPARENT)
                     = (0x50000000, 0x3c, NOPARENT);

7. 4000193c: st %g1, [%fp + -8]
   (a) addr_oid = TID2OID[#fp_tid];
   (b) OID2BaseBound[addr_oid] = IGNORE_CHECK
   (c) The boundary checking for the frame pointer is ignored because source code instrumentation enforces that the frame pointer can only be used for memory access that is guaranteed safe. All unsafe memory accesses should use other registers as the address. In this example, the accesses to the variable *a (stored at [%fp + -8]) and *b (stored at [%fp + -4]) are safe because their addresses are never taken.
   (d) MEM2ID[%fp + -8] = TID2OID[#1] = #1;

8. 40001940: ld [%fp + -8], %g1
   (a) %g1.tid = tid_gen() = #2;
   (b) addr_oid = TID2OID[#fp_tid];
   (c) OID2BaseBound[addr_oid] = IGNORE_CHECK
   (d) The boundary checking for the frame pointer is ignored.
   (e) TID2OID[#2] = MEM2OID[%fp + -8] = #1;

9. 40001944: add %g1, 0x50, %g1;
   (a) %g1.tid = %g1.tid = #2
   (b) %g1 = %g1 + 0x50 = 0x50000050;

10. 40001948: st %g1, [%fp + -4]
    (a) addr_oid = TID2OID[#fp_tid];
    (b) OID2BaseBound[addr_oid] = IGNORE_CHECK
    (c) The boundary checking for the frame pointer is ignored.
    (d) MEM2ID[%fp + -4] = TID2OID[#2] = #1;

11. 4000194c: ld [%fp + -4], %g1
    (a) %g1.tid = tid_gen() = #3;
    (b) addr_oid = TID2OID[#fp_tid];
    (c) OID2BaseBound[addr_oid] = IGNORE_CHECK
    (d) The boundary checking for the frame pointer is ignored.
    (e) TID2OID[#3] = MEM2OID[%fp + -4] = #1;

12. `40001950:  ld [ %g1 ], %g1`
    (a) `addr_oid = TID2OID[%g1.tid] = TID2OID[#3] = #1;`
    (b) `(base, bound, parent_oid)`
        `= OID2BaseBound[#1] = (0x50000000, 0x3c, NOPARENT)`
    (c) `parent_oid == NOPARENT`, so no need to check parent object.
    (d) **Since `%g1 = 0x50000050` is not within `(0x50000000, 0x5000003c)`, so the security engine raises an exception.**

From the step-by-step execution, we can see that the *first pointer* to the allocated buffer is created and put into the `%g1` register at Steps 3-6. During the execution, even if the pointer in `%g1` is stored to the memory (Step 7 and Step 10), its association with its pointed object is not lost. The AHEMS can recover the association when the pointer is loaded back to the register `%g1` using the `MEM2OID` and `TID2OID` table (Step 8 and Step 11). Finally, AHEMS successfully detects the violation of memory safety in `outbound.c` at Step 12.

```
40001924 <main>:
#include <stdio.h>
#include <stdlib.h>
int main()
40001924: 9d e3 bf 98  save %sp, -104, %sp
  int *a = (int *)malloc(sizeof(int) * 15);
40001928: 90 10 20 3c  mov 0x3c, %o0
4000192c: 40 00 00 22  call 400019b4 <malloc>
40001930: 01 00 00 00  nop
40001934: 82 10 00 08  mov %o0, %g1
40001938: 83 b8 40 22  alloc %g1, 0x3c, %g1
4000193c: c2 27 bf f8  st %g1, [ %fp + -8 ]
  int *b = a + 20;
40001940: c2 07 bf f8  ld [ %fp + -8 ], %g1
40001944: 82 00 60 50  add %g1, 0x50, %g1
40001948: c2 27 bf fc  st %g1, [ %fp + -4 ]
  printf("%08X\n", *b);
4000194c: c2 07 bf fc  ld [ %fp + -4 ], %g1
40001950: c2 00 40 00  ld [ %g1 ], %g1
40001954: 05 10 00 2d  sethi %hi(0x4000b400), %g2
40001958: 90 10 a0 70  or %g2, 0x70, %o0
4000195c: 92 10 00 01  mov %g1, %o1
40001960: 40 00 02 1e  call 400021d4 <printf>
40001964: 01 00 00 00  nop
  return 0;
40001968: 82 10 20 00  clr %g1
}
4000196c: b0 10 00 01  mov %g1, %i0
40001970: 81 e8 00 00  restore
40001974: 81 c3 e0 08  retl
40001978: 01 00 00 00  nop
```

Figure 4.11: Disassembly of the Example `outbound` in SPARC

CHAPTER 5

IMPLEMENTATION ISSUES

In this chapter, the pros and cons of the three possible placements of AHEMS are discussed. Following that, the implementation details of AHEMS on the main processor, the runtime monitor, and the security engine are described.

5.1 Placement of AHEMS

Due to AHEMS' asynchronous operation with respect to the main processor, the security engine can be flexibly implemented as an external device, as a co-processor, or as part of the main processor. Below we discuss the pros and cons of each option.

5.1.1 External Device Design

Figure 5.1 shows the architecture of AHEMS where the security engine is implemented as an external device (e.g., a PCI Express card) pluggable to the motherboard. The runtime monitors still need to reside in the cores of the main processor to keep track of every memory event (i.e., loads, stores, allocations, deallocations, and sub-object creations). The security engine

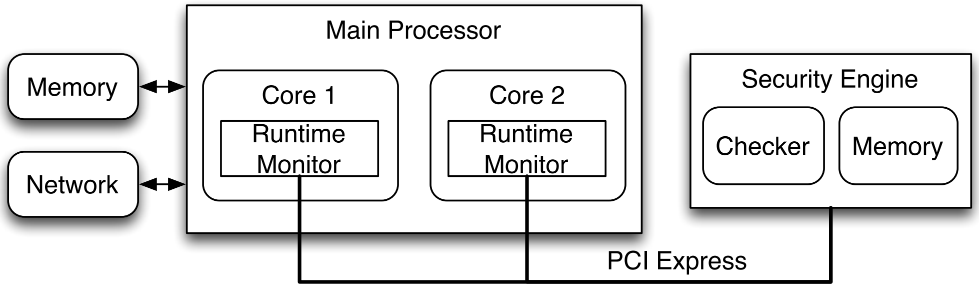


Figure 5.1: External Device Design of the AHEMS

receives memory events from the runtime monitor through the PCI Express Bus, which can reach as high as 31.51GB/s throughput when using the PCI Express 4.0 with 16 lanes. The security engine needs to have its own memory to maintain `MEM2OID`, `TID2OID`, and `OID2BaseBound` tables. To show that PCI Express 4.0 is fast enough to transmit the memory events coming from the runtime monitor, the following is an estimation of the size of the events. If a processor runs at 3GHz with an average IPC $= 2$ (Instructions Per Cycle) and 40% of the executed instructions are `alloc`, `dealloc`, `subcreate`, `load`, `store` instructions, then the expected size of memory events per second would be 3 GHz × 2 instrs/cycle × 40% * 8 bytes/event $=$ 19.2GB/s, which is transmittable by PCI Express 4.0.

The advantages of an external-device design are (1) the changes on the main processor are the least among the three designs because only the runtime monitor needs to be placed in the main processor, (2) the security engine has a separate memory on the PCI card so that the metadata is physically separated from attackers and the security engine does not need to share the memory bandwidth with the main processor, (3) the security engine can be easily upgraded or replaced if there are fixes for bugs or updates for the new features, and (4) the security engine may be produced by different vendors with different features and performance variations.

The disadvantages are (1) the data transmission rate of PCI Express is the slowest among the three designs, so the detection latency and performance overhead are higher, (2) the power consumption is high because the data transmission is going on all the time, (3) the resource overhead is the highest among the designs because the security engine needs additional hardware and memory, and (4) using the PCI Express Bus to communicate information on memory events may compete for the bandwidth with other devices that use the PCI Express to communicate, such as GPUs.

5.1.2 Co-processor Design

Figure 5.2 illustrates the co-processor design of AHEMS architecture. It is similar to the design for an external device except that in the co-processor design, the security engine is a co-processor on the motherboard and the communication between the main processor and the co-processor is inter-

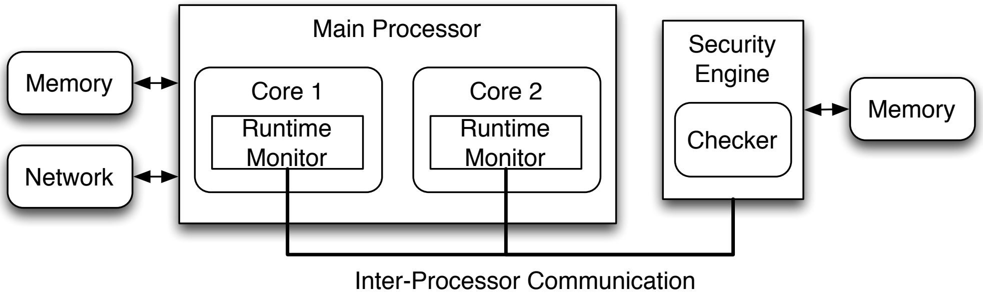


Figure 5.2: Co-processor Design of AHEMS

processor communication. Also, the security engine can optionally have a dedicated memory to store metadata.

The advantages of the co-processor design are (1) the changes on the main processor are smaller than for the in-processor design, (2) the co-processor can optionally have a dedicated memory to increase memory bandwidth and support physical separation of program data and security metadata to prevent attackers from targeting AHEMS, (3) the performance overhead and detection latency are lower than in the external device design because the data transmission rate can be higher.

The disadvantages are (1) the power consumption is still higher than in the in-processor design, (2) the resource overhead is similar to that of the external-device design, and (3) using inter-processor communication channels to communicate information on memory events may compete with other processors that use the channel.

5.1.3 In-processor Design

Figure 5.3 shows the in-processor design of AHEMS. It integrates the whole security engine into the cores of the main processor so that no data transmission is needed and the security engine directly performs the memory safety checking in the main processor.

The advantages of in-processor design are (1) the power consumption is the lowest among these three designs because no data transmission is needed, (2) the resource overhead is the lowest because no additional hardware or memory is needed, (3) the performance overhead and the detection latency are lower compared with the external-device design, and (4) there is more

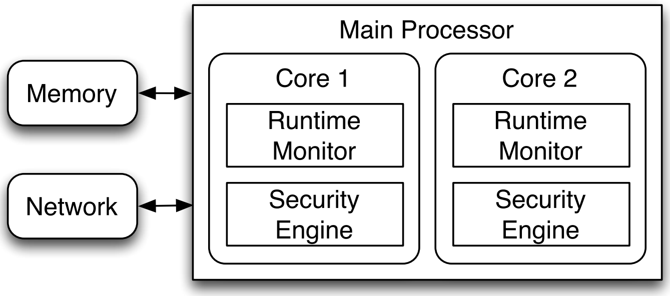


Figure 5.3: In-processor Design of the AHEMS

efficient support for multi-core processors, since each core can have its own security engine, i.e., one security engine does not need to process metadata for different cores.

The disadvantages are (1) the security engine needs to share the memory bandwidth with the main processor, which slows down the security engine, (2) the design requires significant changes to the main processor since it embeds the whole security engine into the main processor, which may affect the critical path of the main processor and thus affect the processor's maximum frequency, (3) the design is less flexible than the external-device design, and (4) the security engine needs to run in the same frequency as the main processor, losing the chance to speed up the checking.

5.1.4 Comparison

Table 5.1 summarizes the differences between the three designs of AHEMS. Each has its advantages and disadvantages. The designer should take into account the system requirements to decide which design option to use based on the tradeoffs involved. In this thesis, we implement the co-processor design to have an average result in all aspects. We use it as a proof-of-concept to demonstrate the effectiveness and efficiency of AHEMS.

5.2 Main Processor

To demonstrate AHEMS in a practical way, instead of using instruction-accurate simulator such as Simics [142], we build AHEMS on top of the open-

Table 5.1: Comparisons of Different Placements of AHEMS

Type	**External Device**	**Co-processor**	**In-processor**
Data Transmission Overhead	High	Moderate	None
Flexibility	High	Moderate	Low
Power Consumption	High	Moderate	Low
Detection Latency	High	Moderate	Low
Resource Overhead	High	Moderate	Low
Changes in Processor	Low	Moderate	High
Effects on Critical Path	Low	Low	High
Physical Separation of Data	Yes	Yes	No
Multi-core Support	Low	Low	High

source Leon3 Architecture developed by Aeroflex Gaisler AB [143]. Leon3 architecture is a full system that includes processors, caches, networks, and other peripherals. The Leon3 processor is a 32-bit processor compliant with SPARC V8 architecture and is a 7-stage-pipeline in-order processor with support for FPU and SRMMU (SPARC Reference Memory Management Unit). Leon3 architecture is a synthesisable VHDL model that is more accurate than any simulator and enables us to even evaluate the critical path, power consumption, and hardware resource overhead of our hardware design.

Figure 5.4 illustrates implementation of AHEMS on the Leon3 architecture. The Leon3 processor communicates with the main memory or network through the bus AMBA 2.0 [144]. The runtime monitor is embedded in the main processor to generate and propagate the TID. It also sends the memory events to the security engine via a FIFO buffer with configurable size. The security engine maintains the metadata in its dedicated memory and checks to see whether any memory event violates memory safety. It raises an exception to notify the runtime monitor if it detects a violation.

5.2.1 TID Propagation.

To support TID propagation, each register in the main processor is widened by $log_2(N)$ bits (i.e., the number of bits per register is increased by $log_2(N)$ bits) to record its associated TID, where N is the total number of registers in the main processor. A TID needs at least $log_2(N)$ bits because, in the worst case, every register points to an object that is not pointed to by any other register, which means each register has a unique TID. Also, objects not

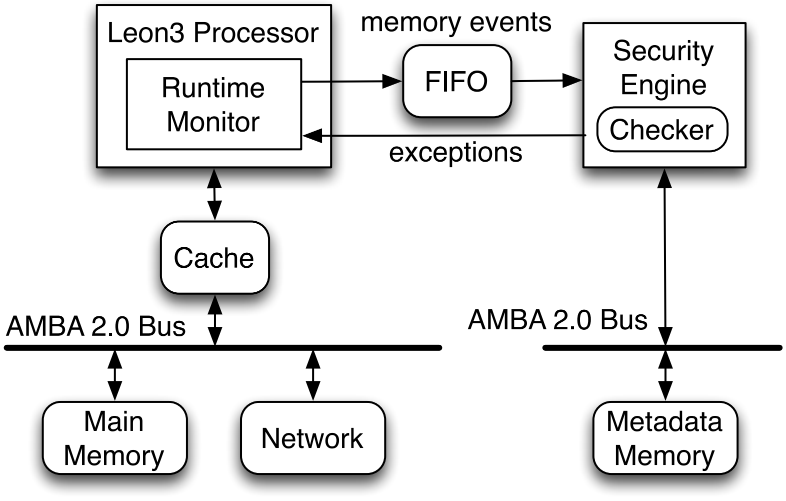


Figure 5.4: AHEMS on Leon3 Architecture

currently pointed to by any register do not need TIDs. Hence, the maximum required number of TID is N. However, a good TID generator (i.e., one which avoids TID collision) is needed to efficiently generate a TID unused by any register at the time of generation. The next section will describe our design of the TID generator.

The ALU (Arithmetic Logic Unit) of the Leon3 processor also needs to be modified to support the propagation of the TID, along with those instructions used for pointer arithmetics. Specifically, the inputs and outputs of the ALU are widened to take the widened registers as inputs and compute a widened result as an output. The resultant TID is computed in the ALU according to the propagation rules in Table 4.1. Since the computation of the TID is in parallel with the original computation, this modification does not require more cycles to run although it may slightly increase the complexity of the ALU.

5.2.2 `alloc`, `dealloc` and `subcreate` Instructions

The `alloc`, `dealloc`, and `subcreate` machine instructions are implemented by reinterpreting the co-processor instructions `cpop1` and `cpop2` in the Leon3 processor. Originally, the co-processor instructions defined by SPARC architecture are used to control the co-processor. Leon3 Processor implements `cpop1` and `cpop2` instructions but treats them as `nop` instruction by default.

```
cpop1 opc, rs1, rs2, rd
cpop2 opc, rs1, rs2, rd
```

Figure 5.5: The Format of cpop1 and cpop2 Instructions

Figure 5.5 shows the format of cpop1 and cpop2. The opc is the opcode given to the co-processor. rs1, rs2, and rd are source register 1, source register 2, and destination register, respectively. We modified the Leon3 Processor to treat the cpop2 with opc=0x01 as the alloc instruction, cpop2 with opc=0x02 as the dealloc instruction, and cpop2 with opc=0x03 as the subcreate instruction. Specifically, the three single-bit signals is_alloc, is_dealloc, is_subcreate are added to the decoder of the Leon3 processor to notify the runtime monitor in the decoding stage of the occurrence of the alloc, dealloc, subcreate instructions, respectively. If one of those signals is set, the runtime monitor will read the data coming from the register file in the *register access* stage. In the case of the alloc and subcreate instructions, the runtime monitor reads the base information from the rs1 register and the bounds information from the rs2 register. In the case of the dealloc instruction, the runtime monitor only reads the pointer to be deallocated from the rs1 register. For the alloc instruction, the runtime monitor also generates a new TID in the *execution* stage, and the ALU of the Leon3 processor is modified to use the base address from rs1 and the new TID as the widened result. In the *exception* stage, the widened result from the ALU is written back to the rd register, and the new *TID* along with base and bound information are sent to the security engine by the runtime monitor through FIFO to notify the allocation event. For the dealloc instruction, no data needs to be written back to the rd register. The runtime monitor directly sends the pointer to be deallocated to the security engine through FIFO in the *exception* stage to notify it of the deallocation event. The handling of the memory allocation and deallocation event by the security engine is described in Section 5.4. The implementation of the subcreate instruction is similar to that of the alloc instruction. However, at this stage of development, our implementation prototype does not support subcreate. We leave it to future work. In the following sections, the implementation of sub-object creation is omitted.

5.3 Runtime Monitor

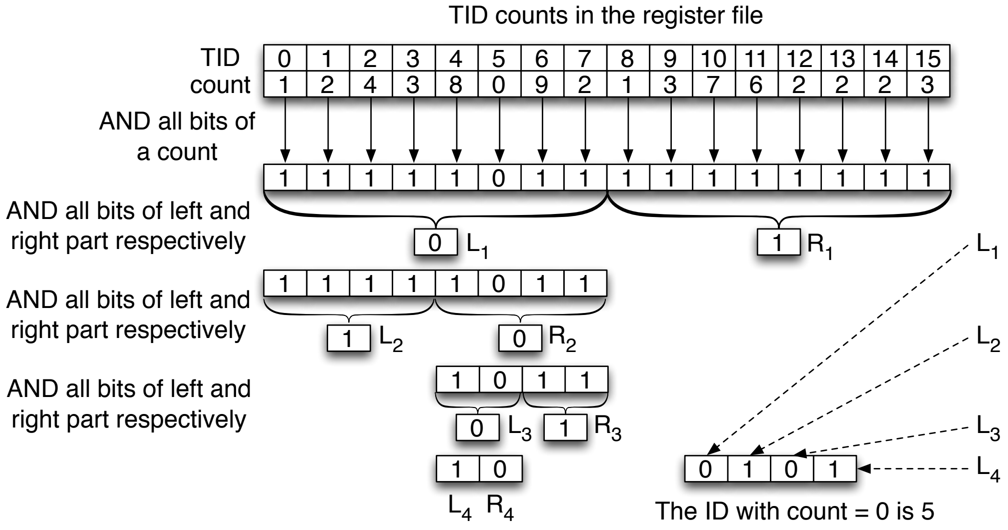
The runtime monitor is responsible for TID generation and data transmission to the security engine. The implementation details are described below.

5.3.1 TID Generation

A new TID is defined as a TID that does not exist in any register in the main processor. It is generated and associated with the destination register `rd` when a `load` or `alloc` instruction is executed. The TID generation process is represented by `id_gen()` in Table 4.1. To generate a new TID efficiently, the TID Generator is developed to keep track of the count of each TID and generate the TID with count = 0. The TID Generator monitors the pipeline of the Leon3 processor to update its `count` table when an old TID value `old_tid` is being overwritten by a new TID value `new_tid` in the register file. Specifically, the TID Generator decrements the count of the TID `old_tid` and increments the count of the TID `new_tid` to maintain the `count` table. With the count of each TID, it uses Algorithm 1, which is similar to a binary search, to find the TID with count = 0. In Algorithm 1, the for-loop in lines 2-4 can be done in parallel, and the loop in lines 6-13 has only

```
   Input: $count[tid]$: the count of the $tid$ in the register file
1         $N$: the total number of registers
   Output: A new TID that does not exist in any register
2  for $tid \leftarrow 0$ to $N-1$ do
3      $iszero[tid] \leftarrow$ (bitwise-AND all bits of $count[tid]$);
4  end
5  $cur \leftarrow iszero$;
6  for $i \leftarrow 0$ to $(\log N - 1)$ do
7      $newtid[i] \leftarrow$ (bitwise-AND the first half of $cur$);
8      if $newtid[i] = 0$ then          // tid w/ count=0 is in 1st half
9          $cur \leftarrow$ the first half of $cur$;
10     else                             // tid w/ count=0 is in 2nd half
11         $cur \leftarrow$ the second half of $cur$;
12     end
13 end
14 return $newtid$
```

Algorithm 1: Generation of a New TID



TID counts in the register file

TID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	1	2	4	3	8	0	9	2	1	3	7	6	2	2	2	3

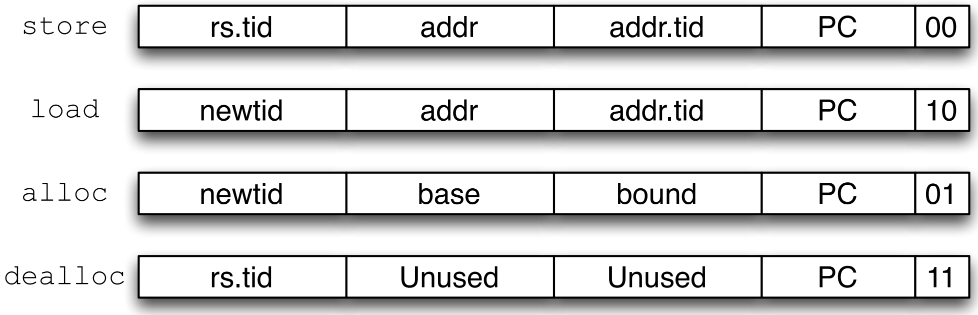
AND all bits of a count
1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
AND all bits of left and right part respectively
0 L1
1 R1
1 1 1 1 1 0 1 1
AND all bits of left and right part respectively
1 L2
0 R2
1 0 1 1
AND all bits of left and right part respectively
0 L3
1 R3
1 0
L4 R4
L1
L2
L3
L4
0 1 0 1
The ID with count = 0 is 5

Figure 5.6: An Example Scenario of the TID Generator when $N = 16$

$\lceil \log N \rceil$ iterations. Therefore, Algorithm 1 can finish in $(\lceil \log N \rceil + 1)$ levels of propagations. In our implementation, N is 136 so the algorithm can finish in 9 levels of propagations, which can fit in one cycle of our implementation. If faster implementation is required, the for-loop in lines 6-13 can be pipelined to achieve higher performance. Figure 5.6 shows a working scenario of the TID Generator with $N = 16$. In the example, TID 5 is the only one whose count is 0. For each *count* of each TID, all bits of the *count* are ANDed to compute the $iszero$ array. The goal of the TID Generator is to find the TID whose $iszero$ entry is 0. After 4 levels of binary search, with each level deciding one bit of the answer, the TID Generator finally computes 5 as the TID with count=0.

5.3.2 Data Transmission

In our implementation, the Leon3 processor is connected to the security engine through a FIFO with configurable size. The size of the FIFO may affect the detection latency of the memory safety violation, which we discuss in Chapter 7. Each entry of the FIFO contains the necessary metadata of a memory event needed by the security engine. Specifically, Figure 5.7 shows the format of the FIFO entry for each memory event in our implementation. When an `alloc/dealloc/store/load` instruction is executed, the runtime



store	rs.tid	addr	addr.tid	PC	00
load	newtid	addr	addr.tid	PC	10
alloc	newtid	base	bound	PC	01
dealloc	rs.tid	Unused	Unused	PC	11

Figure 5.7: Format of FIFO Entry for each Memory Event: `newtid` is the TID generated by TID Generator. `rs.tid` is the TID associated with the source register of the `store` instruction. `addr` is the address value of the `load` or `store` instruction. `addr.tid` is the TID associated with the address. `PC` is the address of the instruction.

monitor pushes a new entry to the FIFO. On the other side, the security engine pops an entry whenever it is ready to check the event. Note that the execution frequency of the main processor does not need to be the same as that of the security engine, which makes asynchronous checking more beneficial. The security engine [1] can typically run faster than the main processor because the functionality of the security engine is much simpler than the main processor's.

5.4 Security Engine

The security engine is responsible for checking the memory safety and maintaining the consistency of the three tables `TID2OID`, `OID2BaseBound`, and `MEM2OID` with the program execution state. The security engine is implemented as a state machine in hardware. Each time the state machine pops out a memory event from the FIFO, the event is processed according to the rules defined in Table 4.1.

[1] In our implementation prototype, we do not make the execution frequency of the security engine different from the main processor's.

Table 5.2: Execution Time of Actions Performed for Each Instruction

Instruction Type	**Execution Time**
`load` instruction	5 cycles + 2 memory accesses
`store` instruction	4 cycles + 2 memory accesses
`alloc` instruction	3 cycles + 1 memory accesses
`dealloc` instruction	4 cycles + 2 memory accesses

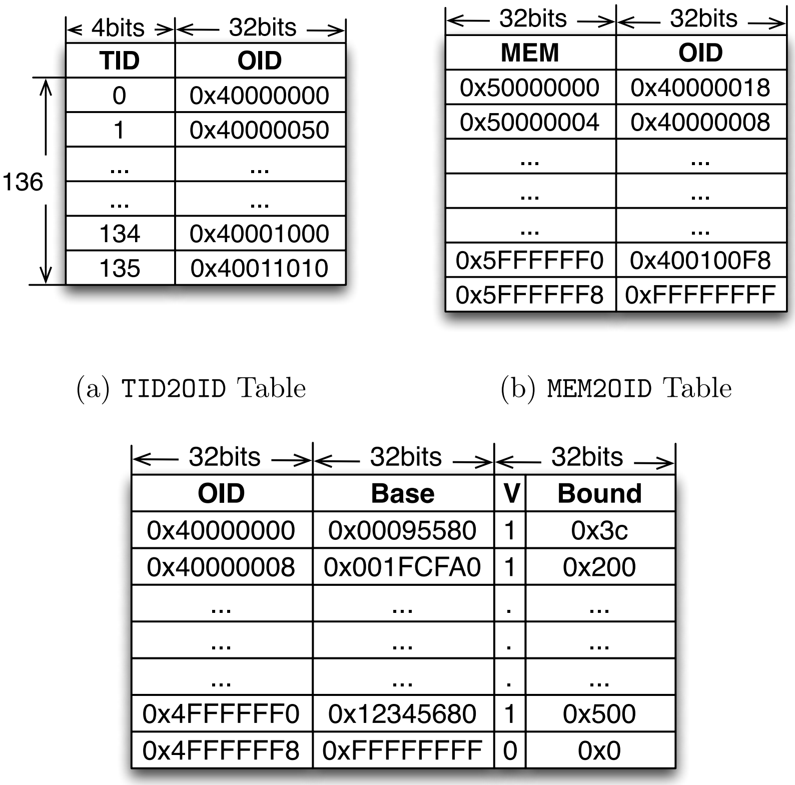
5.4.1 Lookup Tables

Figure 5.8 shows the layout of each table. In the following, we describe the implementation of these three tables.

`TID2OID` table. In our implementation, there are 136 registers in the main processor. Therefore, the `TID2OID` needs 136 entries, and each entry contains a 32-bit wide OID. In total, the `TID2OID` table needs $136 \times 4 = 544$ bytes of storage space. Since the `TID2OID` table can fit in the size of a register file, it is implemented as a register file to speed up the lookup process.

`OID2BaseBound` table. The number of objects in a program can easily exceed the limit of a register file. Therefore, the `OID2BaseBound` table needs to reside in the dedicated memory of the security engine. The `OID`, which is defined to be 32 bits wide in our implementation, can be directly used as the address of the dedicated memory to locate the `(base, bound)`. Each entry of the `OID2BaseBound` table is 8 bytes with 4 bytes each for base and bound, respectively. We currently employ a naive OID Generator, which uses a counter starting at the address `0x40000000` as the new OID and increments the counter by `0x8` each time a new OID is generated. The MSB (Most Significant Bit) of the `bound` is used to indicate whether this entry is valid.

`MEM2OID` table. The `MEM2OID` table also resides in the dedicated memory ranging from `0x50000000` to `0x5FFFFFFF`, with each entry containing an OID that is 32 bits wide. In our implementation, when the security engine uses an address `addr` to lookup the `MEM2OID` table, it first replaces the most significant nibble of `addr` with `0x5`. It then uses the modified address as the index of the `MEM2OID` to fetch the OID, so that the index is always within `(0x50000000, 0x5FFFFFFF)`. For example, if the `addr` is `0x70095580`, this is translated to `0x50095580` for an index. To avoid collisions between addresses, one can employ an MMU to translate the `addr` into the index of the `MEM2OID` table. We leave the use of MMU to future work.


4bits
32bits
136
32bits
32bits
32bits
32bits
32bits

TID	OID
0	0x40000000
1	0x40000050
...	...
...	...
134	0x40001000
135	0x40011010

(a) `TID2OID` Table

MEM	OID
0x50000000	0x40000018
0x50000004	0x40000008
...	...
...	...
...	...
0x5FFFFFF0	0x400100F8
0x5FFFFFF8	0xFFFFFFFF

(b) `MEM2OID` Table

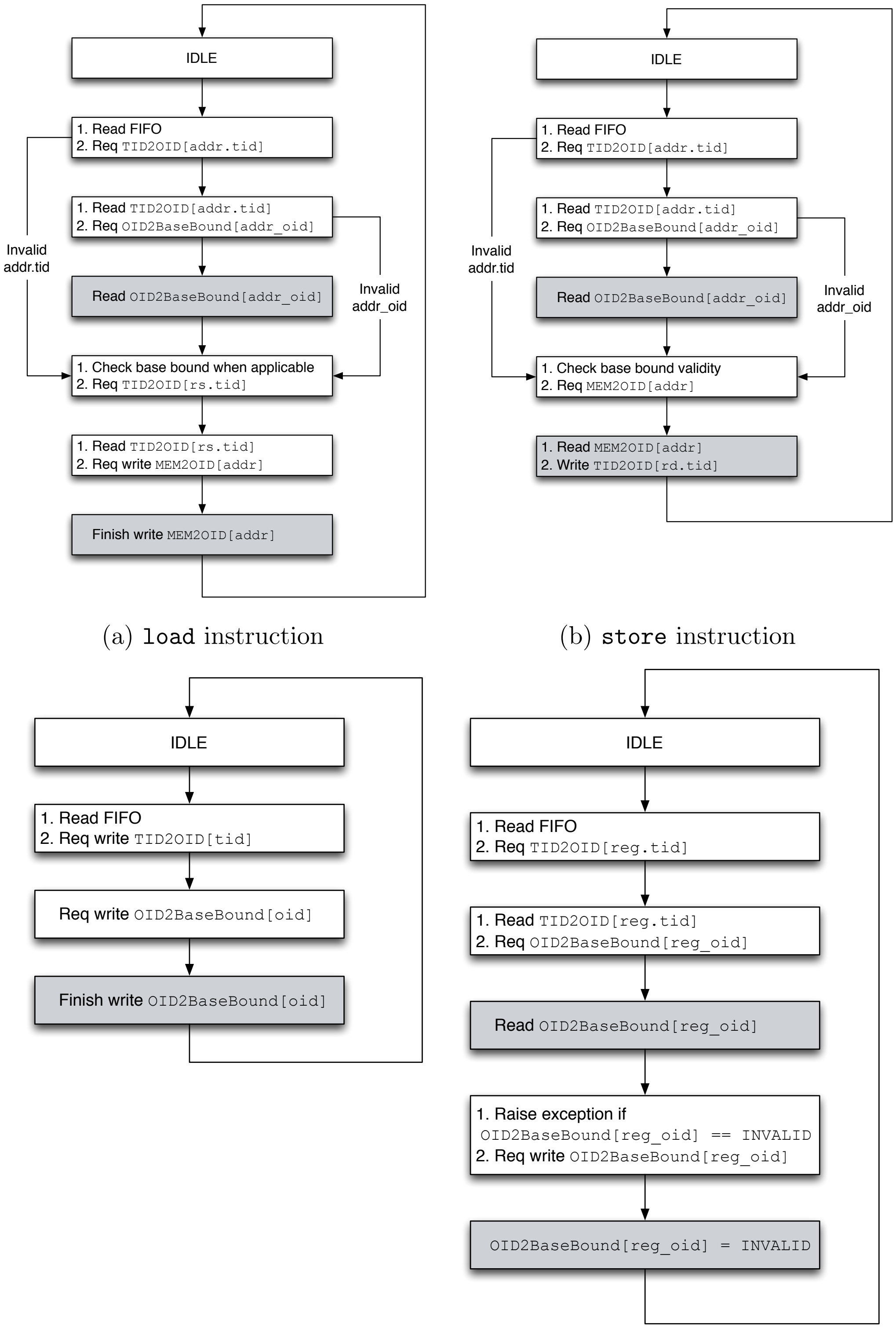
OID	Base	V	Bound
0x40000000	0x00095580	1	0x3c
0x40000008	0x001FCFA0	1	0x200
...	...	.	...
...	...	.	...
...	...	.	...
0x4FFFFFF0	0x12345680	1	0x500
0x4FFFFFF8	0xFFFFFFFF	0	0x0

(c) `OID2BaseBound` Table

Figure 5.8: Layout of Lookup Tables

5.4.2 State Machine

The actions performed along with each `alloc/dealloc/load/store` instruction are implemented by hardware state machines in the security engine. Some actions require memory access, while others require only register file access. Figure 5.9 illustrates the state machine for each instruction. Each white box takes one cycle to finish, while each gray box takes the time of a memory access to finish. For example, actions performed along with a `load` instruction take 5 cycles + 2 memory accesses to finish. Table 5.2 summarizes the execution time of actions performed for each instruction.


IDLE
1. Read FIFO
2. Req TID2OID[addr.tid]
1. Read TID2OID[addr.tid]
2. Req OID2BaseBound[addr_oid]
Read OID2BaseBound[addr_oid]
1. Check base bound when applicable
2. Req TID2OID[rs.tid]
1. Read TID2OID[rs.tid]
2. Req write MEM2OID[addr]
Finish write MEM2OID[addr]
Invalid addr.tid
Invalid addr_oid
IDLE
1. Read FIFO
2. Req TID2OID[addr.tid]
1. Read TID2OID[addr.tid]
2. Req OID2BaseBound[addr_oid]
Read OID2BaseBound[addr_oid]
1. Check base bound validity
2. Req MEM2OID[addr]
1. Read MEM2OID[addr]
2. Write TID2OID[rd.tid]
Invalid addr.tid
Invalid addr_oid
IDLE
1. Read FIFO
2. Req write TID2OID[tid]
Req write OID2BaseBound[oid]
Finish write OID2BaseBound[oid]
IDLE
1. Read FIFO
2. Req TID2OID[reg.tid]
1. Read TID2OID[reg.tid]
2. Req OID2BaseBound[reg_oid]
Read OID2BaseBound[reg_oid]
1. Raise exception if OID2BaseBound[reg_oid] == INVALID
2. Req write OID2BaseBound[reg_oid]
OID2BaseBound[reg_oid] = INVALID

(a) `load` instruction

(b) `store` instruction

(c) `alloc` instruction

(d) `dealloc` instruction

Figure 5.9: State Machines in the Security Engine

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, we experimentally assess the detection coverage of AHEMS and evaluate the runtime performance overhead, hardware resource overhead, power consumption, and critical path on our implementation prototype. In addition, results are compared with prior approaches to show that AHEMS achieves high attack coverage and low performance overhead.

6.1 Experimental Setup

To evaluate the effectiveness and efficiency of AHEMS, we have implemented a prototype of the co-processor-version AHEMS (see Section 5.1.2) on top of the open-source Leon 3 processor [143], which is a synthesizable VHDL design. The prototype is synthesized and run on the FPGA board Xilinx Virtex-5 FXT ML510 [145] with 80 MHz CPU frequency and two DIMMs of 512 MB DDR2 Memory (one as the main memory for the main processor and the other as the metadata memory used by the security engine). Table 6.1 summarizes the details of the experimental setup. For source code instrumentation, the CIL compiler is used to instrument the source code with `alloc()` and `dealloc()` functions so that each memory event is tracked. At this stage of research, our instrumentation has a few limitations: (1) it does not instrument global or static variables, and (2) it does not support the detection of sub-object overflows. However, these are not the limitations of a full-fledged AHEMS (recall the descriptions in Chapter 4 for how we handle global/static variables and sub-objects). Our implementation prototype only works as a demonstration of the efficacy and efficiency of AHEMS.

Table 6.1: AHEMS on Xilinx Virtex-5 ML510 Platform

FPGA Platform	
Model Name	Xilinx Virtex-5 ML510
FPGA	XC5VFX130T-2FFG1738CES
Memory	Two 72-bit 512MB DDR2 RAM (one for the main processor and the other for the metadata memory)
Ethernet	Two on-board 10/100/1000 Ethernet PHYs (MII/RGMII and SGMII)
AHEMS System in FPGA	
GRLIB Version	1.1.0 - B4113
ISA Architecture	SPARC V8
CPU	Leon3 7-stage in-order processor
ALU	One Hardware Multiplier & Divider, No FPU
CPU Frequency	80 MHz
Memory Frequency	160 MHz
AHEMS FIFO Size	16 bytes × 1024 entries
Register File	136 32-bit registers, 8 windows
L1 Data Cache	16KB, 4-way associativity, 16-byte line
L1 Instruction Cache	32KB, 4-way associativity, 32-byte line
Instruction TLB	8 entries, fully-associative
Data TLB	8 entries, full-associative
Operating System	Snapgear Linux 2.6.21.1
Compiler	sparc-linux-gcc 3.4.4
Synthesis Tool	Xilinx ISE 13.4

6.2 Detection Coverage

AHEMS ensures both spatial and temporal memory safety, which prevents most memory corruption attacks. Specifically, AHEMS not only detects control-data attacks, non-control-data attacks, and code reuse attacks by checking the address of each store instruction, but also detects information leakage attacks by checking the address of each load instruction. To evaluate the detection coverage of our AHEMS prototype, we employ the Juliet Test Suite Version 1.2 [146] provided by NIST (National Institute of Standards and Technology). The Test Suite contains a total of 61,387 test cases from 118 different CWEs (Common Weakness Enumerations) for C/C++. Since (1) not all of the test cases are related to memory corruption attacks, (2) there are many similar test cases, and (3) the testing cannot be automated (AHEMS prototype stalls the main processor when a violation is detected), we assess the AHEMS prototype by running only representative test cases from each of the CWEs that lead to memory corruption attacks. We select only those test cases different enough from one another as our representative test cases. Note that those test cases that are written in C++ or that use `ALLOCA()` to initialize local variables are not included because our prototype does not support C++ and the instrumentation of `ALLOCA()` function. Table 6.2 summarizes the test cases we run on AHEMS. In Table 6.2, the first 8 CWEs are caused by spatial memory errors, while the last 3 CWEs are caused by temporal memory errors. These 10 CWEs cover most of the situations that lead to memory corruption attacks. In particular, these CWEs assess defenses against stack-based overflow attacks, heap-based overflow attacks, integer overflow attacks, use-after-free attacks, double-free attacks and dangling-pointer attacks. We have tested a total of 677 test cases with only one case undetected. The undetected test case overflows a sub-object in a structure to overwrite the pointer in the next field of the structure (see Chapter 7 for more details). Although the detection of sub-object overflow is currently not supported by our AHEMS prototype, it is not a limitation of AHEMS.

Compared with other approaches, AHEMS has better coverage on the CWE562 Return of Stack Variable Address, which is a temporal memory error also known as the dangling-pointer vulnerability. Our AHEMS prototype can catch the temporal violations caused by the two cases in the

```
 1 static char *helperBad() {
 2   char charString[] = "helperBad string";
 3   /* FLAW: returning stack-allocated buffer */
 4   return charString;
 5 }
 6
 7 void CWE562_bad() {
 8   // printLine() takes a string pointer as input and
 9   // prints out the string
10   printLine(helperBad());
11 }
```

Figure 6.1: A Test Case in CWE562: Return of Stack Variable Address

Juliet Test Suite, whereas other state-of-the-art memory safety tools, including SoftBound+CETS (revision 183880) [89, 90], SAFECode (revision 183880) [81, 82], AddressSanitizer (in clang v3.2 release) [84], mudflap (in GCC 4.4.7) [79], and memcheck (in valgrind v3.7) [147], are unable to detect the two cases of CWE 562. Figure 6.1 shows a snippet of one of the CWE562 test cases. In Figure 6.1, using the stack buffer `charString` as the parameter of `printLine()`[1] after `helperBad()` returns is a temporal memory error because the memory allocated for `charString` is not preserved once the program is outside the scope of `helperBad()`. However, SoftBound+CETS, mudflap, AddressSanitizer and memcheck miss the error and print out a garbage string in line 10 of Figure 6.1. SAFECode also does not raise an exception to this error, although the program prints out the correct string in line 10. One guess we make about this behavior is that the buffer happens to not be overwritten by subsequent uses of the stack. Unlike all of the above approaches, AHEMS successfully detects both of the CWE562 test cases.

6.3 Runtime Performance Overhead

In this section, we evaluate our prototype on the Olden benchmarks [148] and compare the runtime overhead of our approach with other related approaches.

[1] `printLine()` function accesses the memory pointed by `charString` inside its function body.

Table 6.2: CWEs from Juliet Test Suite Tested on AHEMS

Spatial Memory Errors			
CWE No.	**Description**	**Tested**	**Detected**
CWE121	Stack-based Buffer Overflow	209	208
CWE122	Heap-based Buffer Overflow	18	18
CWE124	Buffer Underwrite	102	102
CWE126	Buffer Overread	145	145
CWE127	Buffer Underread	33	33
CWE588	Attempt to Access Child of Non-structure Pointer	34	34
CWE680	Integer Overflow to Buffer Overflow	38	38
CWE761	Free Pointer Not at Start of Buffer	38	38
Subtotal		617	616
Temporal Memory Errors			
CWE No.	**Description**	**Tested**	**Passed**
CWE415	Double-free	38	38
CWE416	Use-after-free	20	20
CWE562	Return of Stack Variable Address	2	2
Subtotal		60	60
Total		677	676

6.3.1 Runtime Overhead on Olden Benchmark

We measure the runtime performance overhead of our AHEMS prototype on Olden benchmarks [148], since the benchmarks contain pointer-intensive programs with an average of 711 LOCs (Lines of Code) and many prior researchers [81–83, 87, 91, 93] have adopted Olden benchmarks for their performance evaluation; this enables the comparison of our approach with prior researches. Since the Olden benchmarks available on the author's website is outdated and thus cannot be directly compiled, we use the SGI version of Olden benchmarks available in the test-suite of the LLVM compiler [149]. There are ten programs in the Olden benchmarks. We are able to instrument those ten programs without any modification to the source code, whereas some approaches still require a significant amount of manual modification for their approaches to work, such as CCured [75] and SafeProc [97]. Figure 6.2 shows the compiler options we use to compile all benchmark programs for AHEMS. `cilly` is the CIL compiler. `--gcc=sparc-linux-gcc` tells the CIL compiler to use `sparc-linux-gcc` as the underlying compiler. `--merge` tells it to merge all files before transformation. `--doAHEMS` tells it to instrument

```
cilly --gcc=sparc-linux-gcc     \
      --merge                   \
      --doAHEMS                 \
      -O3 -static               \
      -msoft-float -mv8         \
      -Wl,--wrap,malloc         \
      -Wl,--wrap,free           \
      -Wl,--wrap,calloc         \
      -Wl,--wrap,realloc        \
      code1.c code2.c ... -o executable
```

Figure 6.2: Compiler Options Used to Compile Programs for AHEMS

the programs with `alloc()` and `dealloc()` functions. `-O3` tells the GCC compiler to optimize the code. `-static` tells it to statically link the library. `-msoft-float` tells it to perform floating points computation without hardware FPU. `-mv8` tells it that the ISA is SPARC V8. `-Wl,--wrap,malloc` tells it to replace the original `malloc()` function with our instrumented `malloc()` functions and so on for `-Wl,--wrap,free`, `-Wl,--wrap,calloc` and `-Wl,--wrap,realloc`. We are able to run all ten programs without any false alarm. Figure 6.3 shows the runtime overhead of our AHEMS prototype on Olden benchmarks. It achieves an average of 10.6% runtime overhead with the maximum overhead being 38.4%.

The benefit of AHEMS is that it can overlap the time spent on memory safety checking with non-memory-access execution time. Because AHEMS is asynchronous to the execution of the main processor, the security engine can have more time to perform checking if the main processor does not send jobs to the security engine (i.e., the main processor is running non-memory operations). For most of the programs in Olden, the security engine overlaps memory safety checking with main processor execution time very well, so the runtime overhead is low. However, for programs that are memory-intensive such as `bisort`, which spends more than 50% of execution time running `Bimerge()` function (a function with high density of memory operations), the security engine may not have enough time to perform memory safety checking. This is because memory safety checking for loads and stores needs two memory accesses while a load/store instruction only needs one. This causes the main processor to be stalled to wait for the security engine because

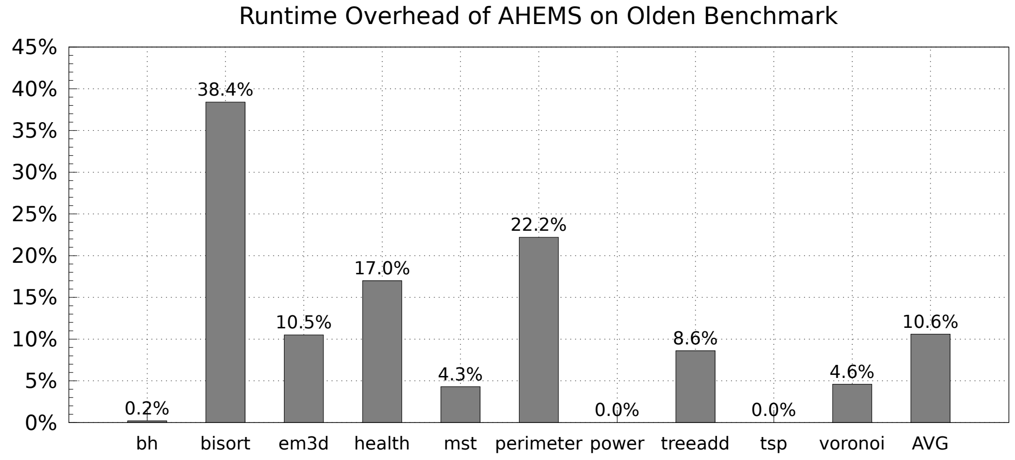


Figure 6.3: Runtime Overhead of AHEMS on Olden Benchmark

Table 6.3: Comparison of Runtime Overhead with Other Approaches

Programs	**Parameters**	**AHEMS**	**Mudflap**	**Softbound +CETS**	**SAFECode**	**Address-Sanitizer**
bh	2000 5	0.2%	13655.2% (false alarm)	– (compile error)	– (runtime error)	41.9%
bisort	2000000	38.4%	3114.0%	341.1%	154.3%	74.7%
em3d	256 250 35 1	10.5%	705.0%	473.0%	192.1%	89.5%
health	9 20 1	17.0%	13343.9%	737.8%	943.2%	361.5%
mst	5000	4.3%	1169.2%	395.1%	644.1%	92.2%
perimeter	11	22.2%	32317.8%	443.1%	212.7%	142.8%
power	None	0.0%	263.9%	1.2%	1.0%	2.7%
treeadd	22	8.6%	106008.1%	448.1%	518.8%	398.7%
tsp	102400	0.0%	1404.9%	206.9%	57.5%	76.8%
voronoi	10000 20 32 7	4.6%	2224.2%	– (false alarm)	– (runtime error)	156.5%
Average		**10.6%**	**17420.6%**	**380.8%**	**340.5%**	**143.7%**

the consumption rate of memory events by the security engine is lower than the production rate of memory events by the main processor. Note that the fact that AHEMS allows the program to continue execution before memory safety checks are done may result in detection latency. The implication of detection latency is discussed in the next chapter.

6.3.2 Performance Comparison

To compare AHEMS with state-of-the-art memory safety approaches, we run the following four approaches on Olden benchmarks: Mudflap, Softbound+CETS, SAFECode, and AddressSanitizer. Mudflap, SAFECode, and AddressSanitizer are object-based approaches, while Softbound+CETS is a pointer-based. We choose these four approaches because: (1) they are publicly available, (2) they have levels of memory safety enforcement similar to AHEMS, and (3) they cover two major categories of memory safety approaches. Since these approaches are software approaches, all runs are performed on a Linux machine with details shown in Table 6.4. Table 6.4 also shows the approach version, compiler version, and compile options used. In Table 6.3, the runtime overhead of our AHEMS prototype is compared with these four approaches. It can be seen that the runtime overhead of AHEMS is an order of magnitude lower than that of any of the other approaches. AHEMS has an average of only 10.6% overhead, while Mudflap, Softbound+CETS, SAFECode and AddressSanitizer have 17420.6%, 380.8%, 340.5%, and 143.7% overheads, respectively. Note that all approaches are run on Olden benchmarks with the same parameters to enable fair comparison. However, the overhead may not be consistent with the overhead reported by the authors of the approaches because: (1) the parameters used may be different, (2) the authors may employ other unreported optimizations, (3) the approaches have been updated since they were published, and (4) the host machine is different.

In comparison with other software approaches with the same level of memory safety enforcement, AHEMS has the lowest runtime overhead on Olden benchmarks. For object-based approaches, J&K [78] has average 5–6x slowdown, while CRED [80] has 11–12x slowdown. For pointer-based approaches, CCured [75] has an average 28% overhead, Patil and Fischer [75] has an average of 538%, MSCC [87] incurs an average of 99% overhead, and MemSafe [91, 92] incurs 29%. Although some software approaches such as WIT [46], BBC [83], and PAriCheck [85] average only 4%, 6%, and 4.5% overheads, respectively, all of them trade off the completeness of memory safety for performance. For example, BBC and PAriCheck do not detect dangling pointers that point to reallocated memory (one kind of temporal memory error), and WIT does not prevent any spatial or temporal memory error that occurs on load instructions. In contrast, AHEMS protects against both spatial and temporal memory errors on all memory accesses.

For hardware approach, Arora et al. [150] have 10% and 100% overhead on `em3d` and `health`, respectively, higher than the respective overheads of AHEMS. SafeProc [97] achieves an average of 9% overhead, which is slightly lower than AHEMS. However, it requires non-trivial source code modification, which may impede its deployment. Although HardBound [93] has an average of 9% overhead when using a 4-bit external pointer encoding scheme and 7% when using a 4-bit internal pointer encoding scheme, it only enforces spatial memory safety. Watchdog [94] extends HardBound to further ensure temporal memory safety, but the authors of Watchdog do not report runtime overhead on Olden benchmarks. Their average runtime overhead on 20 SPEC benchmarks is 24%. Clause et al. [95, 96] do not report overhead on Olden benchmarks. Their runtime overhead on SPEC benchmarks ranges from 5% to 21% depending on the number of used taint marks. Increasing the taint marks increases the overhead. However, Clause et al. suffer from false positives and false negatives if the number of taint marks is too small. Chuang et al. [98] have an average of 21.2% overhead on SPEC benchmark. Despite the fact that SafeMem [99] has an average of 6.3% overhead on non-standard benchmarks and MemTracker [100, 101] has 2.7% on SPEC benchmarks, their protections on memory safety are much less comprehensive than AHEMS. For example, SafeMem does not detect attacks that allow arbitrary memory writes, while MemTracker fails to detect buffer overflow that overwrites decision-making variables in the stack to launch non-control data attacks.

In summary, AHEMS has very low performance overhead compared with other software and hardware approaches of the same level of memory safety enforcement. For those that have overhead slightly lower than AHEMS, they either have incomplete protection on memory safety or have false positives and false negatives compared to AHEMS.

6.4 Memory Overhead

Our AHEMS prototype uses one dedicated DIMM of memory to store base and bounds metadata. Therefore, our prototype has a fixed amount of memory overhead over all programs. In our case, the memory overhead is 512MB. The benefit of a dedicated memory is that it provides higher memory bandwidth and physical isolation for the metadata. If one DIMM of memory is too much overhead, the system designer can also arrange for AHEMS to store metadata in the same memory as the main processor, with the tradeoff in terms of memory bandwidth and metadata protection.

Table 6.4: Experimental Environment for the Four Approaches

Host Machine	
CPU	Intel Core i5-3470 CPU @ 3.40GHz x 4
Memory	2 x 4 GB DDR3 1600MHz RAM
OS	Ubuntu 12.04 LTS
Mudflap	
Version	Mudflap module in GCC 4.4.7
Compiler	GCC 4.4.7
Compile Options	-g -O3 -fmudflap -lmudflap
Softbound+CETS	
Softbound+CETS Version	safecode revision 183880
Compiler	clang v3.2 release (r183906)
Compile Options	-g -O3 -fsoftbound
SAFECode	
SAFECode Version	safecode revision 183880
Compiler	clang v3.2 release (r183906)
Compile Options	-g -O3 -flto -use-gold-plugin -fmemsafety
AddressSanitizer	
AddressSanitizer Version	clang v3.2 release (r183906)
Compiler	clang v3.2 release (r183906)
Compile Options	-g -O3 -fsanitize=address

6.5 Critical Path

The critical path of a processor is crucial in determining the maximum frequency of the processor. In other words, the critical path affects the runtime performance overhead. However, prior work on hardware approaches seldom estimates the impact of the approach on the critical path of the main processor. The reason is that most of the previous studies use simulator that does not model the critical path. In contrast, we implement AHEMS on FPGA in RTL level so that the impact on the critical path can be estimated. The Leon3 Processor running on Xilinx Virtex-5 ML510 has a default execution frequency of 80 MHz. Although our AHEMS implementation prototype adds the security engine and the runtime monitor to the Leon3 System, the modified Leon3 System is still able to work with the same frequency of 80 MHz without breaking any timing constraints. Specifically, we synthesize the Leon3 System with and without AHEMS to compare their post-place-and-route static timing using the Xilinx ISE tool. The delay of the critical path with and without AHEMS is 12.470 ns (equivalently, 80.192 MHz) and 12.463 ns (equivalently, 80.238 MHz), respectively. This shows that AHEMS poses negligible impact on the critical path of the Leon3 System (only 0.06% increase) and thus does not affect the performance of the main processor.

6.6 Hardware Resource Overhead

We estimate the hardware resource overhead of our AHEMS implementation prototype by comparing the FPGA hardware utilizations of the Leon3 System synthesized with and without AHEMS enhancement. Table 6.5 lists the hardware overhead on each type of FPGA resource. The definition of Slice in Virtex-5 FPGA is a collection of 4 flip-flops, 4 6-input LUTs, wide-function multiplexers, and carry logic. Therefore, the increased number of Slice Registers, Slice LUTs, and LUT Flip Flop pairs, which are 20%, 25.1%, and 19.3%, respectively, represent the additional logic used to implement the security engine, the runtime monitor, and the metadata propagation in the pipeline. In addition, the number of occupied Slices, which increases by 13.8%, represents the additional Slices needed by AHEMS. However, these Slices are not necessarily fully utilized (i.e., Some flip-flops and LUTs in a

Table 6.5: Device Utilization of AHEMS on Xilinx Virtex-5 ML510

Name	**Leon3 w/o AHEMS**	**Leon3 w/ AHEMS**	**Overhead**
Number of Slice Registers	13323	15988	20.0%
Number of Slice LUTs	22443	28079	25.1%
Number of Route-thrus	450	625	38.9%
Number of Occupied Slices	10730	12211	13.8%
Number of LUT Flip Flop pairs Used	27892	33279	19.3%
Number of Bounded IOBs	440	440	0.0%
Number of BlockRAM/FIFO	33	40	21.2%
Total Memory Used	1062 KB	1278 KB	20.3%
Number of BUFG/BUFGCTRLs	21	21	0.0%
Number of IDELAYCTRLs	6	6	0.0%
Number of BSCANs	2	2	0.0%
Number of DCM_ADVs	7	7	0.0%
Number of DSP48Es	4	4	0.0%
Average Fanout of Non-clock Nets	4.38	4.6	3.0%

Slice are unused). Finally, the increased number of BlockRAM/FIFO and total memory used, which are 21.2% and 20.3%, respectively, are due to the use of FIFO, TID2OID table, and widened register file in the AHEMS prototype. Since the Leon3 System is a relatively simple architecture, the hardware resource overhead may seem high. However, the absolute hardware usage of AHEMS is small compared to commercial processors such as Intel processors. Note that this is a rough estimation, since the synthesis result for FPGA may be different from the synthesis result for ASIC design. To get more accurate results, we can port our AHEMS design from Xilinx ML-510 to the ASIC version of the Leon3 System and synthesize it with an ASIC design compiler, such as Synopsis Design Compiler [151], to get more accurate hardware resource usage data, such as gate counts. We leave this to future work.

6.7 Power Consumption

To estimate the power consumption overhead of our AHEMS prototype, we synthesize the Leon3 System with and without AHEMS enhancement and analyze the designs with the Xilinx XPower Analyzer. Table 6.6 summarizes the power consumption for each component of the synthesized design. From the table, we can see that AHEMS has 94.3% and 22.3% overhead on signals

Table 6.6: On-Chip Power Consumption Overhead of AHEMS

On-Chip	**Leon3 w/o AHEMS**	**Leon3 w/ AHEMS**	**Overhead**
Clocks	421.33 mW	447.11 mW	6.1%
Logic	12.36 mW	15.11 mW	22.3%
Signals	15.32 mW	29.77 mW	94.3%
IOs	6188.84 mW	6187.63 mW	0.0%
BRAMs	48.60 mW	51.58 mW	6.1%
DCMs	523.73 mW	523.73 mW	0.0%
DSPs	0.00 mW	0.01 mW	0.0%
Quiescent	2630.21 mW	2632.40 mW	0.1%
Total	**9839.99 mW**	**9887.34 mW**	**0.5%**

and logic, respectively. The overhead from signals is mostly due to the use of flip-flops in the TID generator to keep the counts of each TID in the register file (see Section 5.3.1). In spite of the high overhead of AHEMS on signals and logic, the total power consumption overhead is only 0.5%, as signals and logic take up a very small portion of the total power consumption. Instead, IOs dominate power consumption. Therefore, the impact of signals and logic is negligible. This measurement shows that AHEMS incurs very little power consumption overhead, which enables AHEMS to be deployed on mobile devices, such as smartphones, to prevent malicious attacks.

CHAPTER 7

QUALITATIVE ANALYSIS

In this chapter, different properties of AHEMS are analyzed to show its strengths and limitations.

7.1 Strengths

In this section, we discuss the following properties of AHEMS to show its strengths: (1) completeness of memory safety, (2) source compatibility, (3) binary compatibility, (4) modularity support, (5) scalability, (6) physical metadata isolation, and (7) FPGA implementation.

7.1.1 Completeness of Memory Safety

AHEMS is a hardware approach that adopts an idea of source code instrumentation similar to that of SoftBound + CETS [89, 90] (see Chapter 8 for more details), which is a state-of-the-art solution and proven to enforce complete memory safety under some assumptions (e.g., the integer-to-pointer casts are all annotated by programmers). Due to the similarity, AHEMS can detect all spatial and temporal memory errors covered by SoftBound + CETS. In addition, AHEMS handles metadata propagation better than SoftBound + CETS so that fewer false positives are generated. In particular, AHEMS can handle the case of dereferencing a pointer even if the pointer is copied by `memcpy()`-like operations, (i.e., byte-by-byte instead of all at once). SoftBound + CETS loses track of the pointer's metadata in that case due to the byte-by-byte operations and generates false alarms. Figure 7.1 shows an example program that causes SoftBound + CETS to raise false alarms, while AHEMS handles the program correctly. The program in Figure 7.1 copies the pointer `b` to the pointer `a` in line 12 using `memcpy-like` function called

```
 1 void my_memcpy(char *to, char *from, size_t size) {
 2   int i;
 3   for (i = 0; i < size; i++) {
 4     to[i] = from[i];
 5   }
 6 }
 7 main() {
 8   int ary[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 9   int input;
10   int i, *a, *b;
11   b = ary;
12   my_memcpy(&a, &b, sizeof(b)); // equivalent to a = b
13   printf("%d\n", a[9]); // SoftBound+CETS raises false alarms
14 }
```

Figure 7.1: An Example Code That Causes SoftBound+CETS to Raise False Alarms

my_memcpy. Because SoftBound + CETS does not know how to propagate the metadata when the pointer is copied byte by byte, the metadata in b is not properly propagated to a. Therefore, SoftBound + CETS will raise a false alarm in line 13, which is in fact a legal access. On the other hand, AHEMS always propagates the metadata that belongs to the pointer, even if one byte of the pointer is loaded from or stored into the memory. This allows AHEMS to propagate the metadata properly.

Finally, AHEMS achieves an order of magnitude lower runtime overhead than SoftBound + CETS by employing hardware to perform metadata propagation on pointer arithmetics and to perform runtime checking on pointer dereferences. AHEMS also enables the runtime checking to be performed asynchronously to the execution of the main program, which greatly reduces the runtime overhead. To sum up, AHEMS has the same level of memory safety enforcement (spatial and temporal memory) as SoftBound + CETS, but AHEMS has fewer false positives and better performance.

7.1.2 Source Compatibility

An approach is source compatible if: (1) the approach does not break the assumptions made by programmers for an existing program and (2) it does

```
 1 // Let's assume a 32-bit architecture
 2 struct S1 {
 3   size_t v1;   // offset 0
 4   void * v2;   // offset 4
 5   size_t v3;   // offset 8
 6 };
 7 struct S2 {
 8   size_t v1;   // offset 0
 9   size_t v2;   // offset 4
10   size_t v3;   // offset 8
11 };
12
13 void func(struct S1 *s1) { // A precompiled function
14   struct S2 *s2 = (struct S2 *) s1;
15   ... = s1->v3;
16   s2->v3 = 5678;
17   // the offset of v3 in S1 and the offset of v3 in S2
18   // are different in fat-pointer approaches
19 }
```

Figure 7.2: An Example Code That is Source and Binary Incompatible If Fat-Pointers Approaches are Used

not require manual modifications or annotations to the source code.

Some approaches, such as fat-pointer technique [74–76] (see Chapter 8) change the memory layout of a program in a programmer-visible way, which may violate an assumption made by a programmer and in turn break the execution of the program. Figure 7.2 shows an example program that is not source compatible if fat-pointer approaches are used. In Figure 7.2, the memory layout of `struct S1` and `struct S2` are assumed to be the same by the programmer of the `func()` function. Therefore, the programmer performs a cast in line 14 to typecast `struct S1` to `struct S2` and then accesses the value of `v3` in line 16, hoping to modify the value of `v3` in `S1`. However, if fat-pointer approaches are applied to `func()`, the memory layout of `S1` will be changed into the structure shown in Figure 7.3. Let's assume the program is compiled for a 32-bit architecture. The offset of `v3` in `S1` is 8 without the use of fat pointers, while the offset of `v3` is 16 when fat-pointer approaches are used, since the pointer `v2` takes up 12 bytes. Hence, the assignment in line 16 of Figure 7.2 corrupts the metadata of

```
struct FAT_PTR {
  void *ptr;
  void *base;
  size_t bound;
};
struct S1 { // memory layout is changed
  size_t v1;  // offset 0
  FAT_PTR v2; // offset 4
  size_t v3;  // offset 16
};
struct S2 { // nothing changed
  size_t v1;  // offset 0
  size_t v2;  // offset 4
  size_t v3;  // offset 8
};
```

Figure 7.3: The Structure of `S1` and `S2` under Fat-pointer Approaches

`v2` instead of modifying the correct value of `v3`. This corruption is caused by the first requirement of source compatibility mentioned above. AHEMS maintains the metadata in its dedicated memory, isolated from user data, so that it does not change any memory layout or break any assumption made by programmers. Also, the source code instrumentation of AHEMS is fully automated. AHEMS uses the CIL compiler to instrument global, static, and local variables to keep track of the allocation and deallocation of those variables. The tracking of heap variables is handled by instrumenting the `malloc()`-family and `free()` functions, which does not need to change from program to program. Therefore, AHEMS is fully source compatible.

7.1.3 Binary Compatibility

Binary compatibility (or backwards compatibility) requires that (1) the protected code can interoperate with unprotected binaries (e.g., a third-party or precompiled library whose source code is unavailable for instrumentation) and (2) no false positive is generated, and partial memory safety can still be enforced on unprotected code.

Some approaches, such as fat-pointer techniques, may also make a program fail to interoperate with precompiled binaries. Let's use the same example as

```
1 int* libf(int *ptr) {
2   int k, *rp;
3   ... = *(ptr + 100);
4   ...
5   rp = ptr + k;
6   return rp;
7 }
```

Figure 7.4: An Example Program That May Cause Outdated Metadata

in Figure 7.2 and assume `func()` is a library function in a precompiled binary that is oblivious to fat-pointer approaches. If a user function protected by fat-pointer approaches calls the unprotected `func()` and passes a pointer that points to an instance of `S1` to `func()` as the argument, `func()` will end up getting the wrong value of `v3` in line 15. This is because the offset of `v3` in `S1` known by the `func()` is 8, while the offset of `v3` in `S1` used by the user function is 16.

Other approaches [46,82,91,92,152] that change the interfaces of functions (e.g., the number of arguments) may also have problems interoperating with the precompiled binaries. This is because a function `f()` may be pointed to by a function pointer `*fp` that is used by a function `g()` in a precompiled binary. In this case, if the interface of `f()` is changed, `g()` is unable to correctly pass the arguments to `f()` when using `*fp` to call `f()`. AHEMS does not change any memory layout or the interface of any function, so it can interface with third-party libraries in precompiled binaries without any problem or manual change.

Furthermore, most of the software-only approaches fail to maintain their metadata during the execution of unprotected code. Thus the metadata become outdated after the execution, which can cause false positives and false negatives. Figure 7.4 shows an example program that may cause software-only approaches to lose track of metadata. Let's assume `libf()` in Figure 7.4 is an unprotected function in a precompiled binary. Software-only approaches cannot propagate the metadata for the pointer arithmetics performed in line 5 of Figure 7.4. Therefore, these approaches must either treat the returned pointer `rp` of `libf()` as an unbounded pointer to remove false positives (pointer-based approaches, see Chapter 8) or risk `rp` going out of bounds (object-based approaches). Either case weakens the security guarantee enforced by these approaches. Furthermore, most of software-only approaches cannot perform bounds checking within unprotected code. Therefore, if an out-of-bounds access occurs in line 3 of Figure 7.4, these approaches do not detect the violation. On the other hand, hardware approaches such as AHEMS can still propagate the metadata associated with the input parameter `ptr`, even in the unprotected code, so that the returned pointer is still associated with the correct metadata and a memory safety violation happening in line 3 of Figure can be caught. 7.4.

In summary, AHEMS has better binary compatibility than any other approach except for dynamic instrumentation, which is extremely slow and inapplicable to real-world programs. The binary compatibility of AHEMS enables its incremental deployment on existing systems.

7.1.4 Modularity Support

An approach is said to have modularity support if it can handle independent modules separately. Modularity support allows users to process each file individually and combine them later to save compilation time. For example, if a user changes only a couple of files in the code base, he or she only needs to compile the modified files into intermediate results and then link the new intermediate results and existing ones to get the working executable. Modularity support is important for an approach to be widely adopted because it saves developers' time.

AHEMS has modularity support. It utilizes the CIL compiler to instrument each source file separately and then uses the GCC compiler to compile the instrumented source files, which produces the final executable.

7.1.5 Scalability

In this thesis, scalability is defined as the size of the source code an approach can efficiently and effectively process. Some approaches such as SAFECode [81, 82] and MemSafe [91, 92] perform whole-program analysis, such as inter-procedural pointer analysis, to reduce runtime overhead. However, they suffer from scalability issues, since whole-program analysis is an expensive operation and may also hurt the support for modularity. The source code instrumentation of AHEMS does not require whole-program analysis. AHEMS only needs to instrument the allocation and deallocation of all automatic variables, which is mostly an intra-procedural transformation except for global and static variables. Although the instrumentation of global and static variables is a global transformation, the transformation is still very efficient.

7.1.6 Physical Metadata Isolation

The integrity of metadata is a problem that is ignored by many approaches. For example, the inline metadata of fat-pointer approaches may be corrupted due to arbitrary type casts, as discussed in Section 7.1.2. Ignoring the integrity of metadata makes a protection scheme lose comprehensive protection. To protect metadata, existing approaches either (1) disallow the use of arbitrary type casts, (2) introduce extra data structures, or (3) put the metadata in the shadow memory. The first method makes an approach source-incompatible, while the second method may incur significant runtime overhead. The third method works only if the shadow memory is perfectly protected (i.e., there is no illegal way to modify the metadata). However, complete memory safety does not guarantee there is no vulnerability in the program or the system. For example, memory safety does not forbid an attacker from allocating an object of one type and using it as another type, which may allow the attacker to manipulate the metadata. To address the metadata integrity problem, AHEMS imposes a physical isolation on the metadata. The metadata of each pointer are stored on a dedicated memory managed by the security engine. The dedicated memory is invisible and inaccessible to the main processor, which strongly guarantees the integrity of the metadata.

7.1.7 FPGA Implementation

Some prior hardware approaches [93, 94, 153] use instruction-accurate simulators, such as Simics [142], to conduct their experiments. Some other hardware approaches [95–97, 100] use cycle-accurate simulators, such as SimpleScalar [154] or SESC simulator [155], to perform more accurate performance evaluations. However, the use of such simulators is unable to evaluate the impact of the proposed hardware on critical paths, hardware resource overhead, and power consumption, which are also crucial to the feasibility and the deployment of the proposed hardware. In contrast, we build an FPGA implementation prototype of AHEMS on real hardware, which shows the practicality of our design and our ability to measure more metrics on the performance and resource overhead of the hardware.

7.2 Limitations

In this section, we discuss the following limitations of AHEMS and the potential solutions to them: (1) detection latency, (2) memory overhead, (3) type unsafety, and (4) mixture of multiple pointers.

7.2.1 Detection Latency

The downside of an asynchronous approach is that memory violation detection may be delayed. The larger the FIFO is, the better the performance of AHEMS is, since AHEMS can overlap more of its processing time with the execution time of the main processor. To analyze the detection latency of AHEMS, we can estimate the detection latencies of AHEMS both when the FIFO is empty (minimum detection latency) and when the FIFO is full (maximum detection latency). Given a processor with AHEMS enabled, let $FIFOSize$ be the number of entries in the FIFO, IPC be the average number of instructions per cycle, $MemLatency$ be the average number of cycles for a memory access (taking into account the effect of caches), and $DetectTime$ be the average number of cycles it takes for the security engine to process an FIFO entry. Since `load` and `store` instructions take up most of the FIFO entries (the other two instructions, `alloc` and `dealloc`, are much less frequent), we use the average time of processing `load` and `store` instructions to approximate $DetectTime$ (see Table 5.2 for the processing time of each

FIFO entry). Therefore, we have

$$\begin{aligned} \mathit{DetectTime} &= ((5 + 2 \times MemLatency) + (4 + 2 \times MemLatency))/2 \\ &= 4.5 + 2 \times MemLatency \qquad (7.1) \end{aligned}$$

When the FIFO is empty and an entry E is being pushed into the FIFO, the entry E is processed immediately and so the detection latency for the entry E in terms of number of instructions is

$$\begin{aligned} \mathit{DetectLatency}_{\mathit{empty}} &= \frac{\mathit{DetectTime}}{IPC} \\ &= \frac{(4.5 + 2 \times MemLatency)}{IPC} \qquad (7.2) \end{aligned}$$

On the other hand, when the FIFO is full and an entry E is being pushed into the FIFO, the detection latency for the entry E in terms of number of instructions is

$$\begin{aligned} \mathit{DetectLatency}_{\mathit{full}} &= \frac{\mathit{DetectTime} \times FIFOSize}{IPC} \\ &= \frac{(4.5 + 2 \times MemLatency) \times FIFOSize}{IPC} \qquad (7.3) \end{aligned}$$

The $MemLatency$ for a processor with L1 and L2 caches can be calculated as follows:

$$\begin{aligned} &MemLatency = \text{L1 Cache Hit Time} \\ &+ \text{L1 Cache Miss Rate} \times \text{L2 Cache Hit Time} \qquad (7.4) \\ &+ \text{L1 Cache Miss Rate} \times \text{L2 Cache Miss Rate} \times \text{L2 Cache Miss Time} \end{aligned}$$

To give a real-world example, we calculate the detection latency of AHEMS on the Intel Core 2 Duo E6400 processor with system specifications [156–158] listed in Table 7.1. By plugging values into Table 7.1, we get

$$MemLatency = 5.715 \text{ (cycles)} \qquad (7.5)$$

$$\mathit{DetectLatency}_{\mathit{empty}} = 16.42 \text{ (instructions)} \qquad (7.6)$$

$$\mathit{DetectLatency}_{\mathit{full}} = 16816.82 \text{ (instructions)} \qquad (7.7)$$

An attacker can only exploit the time windows caused by the detection

Table 7.1: System Specifications of an Intel Core 2 Duo Processor Running on the SPEC2006 Benchmark

Attribute	**Value**
CPU	Intel Core 2 Duo E6400 (2 x 2.13GHz)
L1 Cache	32 KB X 2, 8 way, 64–byte cache line size, write-back
L2 Cache	2MB shared cache (2MB x 1), 8-way, 64-byte line size, non-inclusive with L1 cache
Memory	2GB (1GB x 2) DDR2 533MHz
FSB	1066MHz Data Rate 64-bit
FSB Bandwidth	8.5GB/s
L1 Cache Hit Time[a]	3 cycles
L2 Cache Hit Time	12 cycles
L2 Cache Miss Penalty	165 cycles
IPC (Instruction Per Cycle)[b]	0.97
L1 Miss Rate	2.25%
L2 Miss Rate	0.41%
AHEMS FIFO Size	1024 entries × 16 bytes

[a] Cache hit time is the time to access a data when the data is already in the cache.

[b] The IPC and cache miss rates of a processor depend on the workload of the executed program. Therefore, we use SPEC2006 benchmarks as the representative programs to approximate the IPC and cache miss rates of real-world programs.

latency to launch an attack. This means that an attacker has an average of 16.42 instructions to launch an attack when the FIFO is empty and an average of 16816.82 instructions when the FIFO is full. In terms of time, 16816.82 instructions take only 7.66 μs to execute, which means that the detection latency is not even sensible by human. On the one hand, some attacks have two stages. The first stage is to exploit the vulnerability of systems and launch a `shell`, and the second stage requires an attacker to perform further inspection to successfully compromise a system. In this case, the attacker does not have enough time to react, since the attack is detected within a second. For example, there are attacks that launch a `shell` with root privilege for an attacker. However, the attacker still needs time to explore the system for at least more than one second to steal sensitive information or cause damage. On the other hand, even if the attack is fully automated, it may still take more than 7.66 μs to finish. To give an idea of how short 7.66

μs is, even a call to `system("")`[1] with an empty string as a command takes more than 1 ms on a machine with the Intel Core i5-3470 3.2 GHz CPU. This means that a `shell` cannot be launched within 7.66 μs. More importantly, the FIFO size is configurable so that system designers can adjust the size of the FIFO based on the tradeoff between performance and the length of detection latency. If the size of the FIFO is set to 1, then AHEMS can detect a memory error in less than 16.42 instructions, which is too small to launch an attack.

7.2.2 Memory Overhead

Since AHEMS uses a dedicated memory to store its metadata, AHEMS provides physical metadata isolation and dedicated memory bandwidth for metadata. The tradeoff is that the amount of memory overhead is fixed no matter how small the executed program is. In our implementation prototype, we dedicate a DIMM of memory to the metadata. Nonetheless, the in-processor design of AHEMS can share memory with the main processor, which trades off the physical metadata isolation and performance for reduced memory usage. System designers can determine which design of AHEMS to use based on the available memory resource on the system, the intended level of security, and the desired performance.

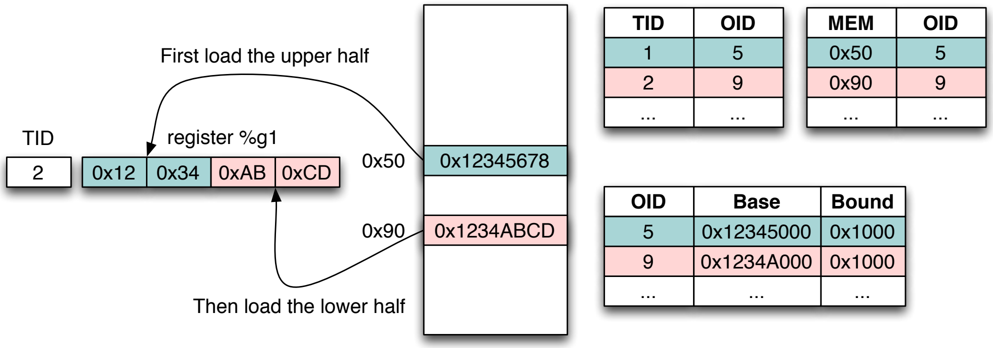
7.2.3 Type Unsafety

AHEMS is designed to enforce memory safety, which prohibits attacks that exploit memory errors. However, AHEMS does not detect some class of type safety violations such as treating one object as another object of equal or smaller size. This error cannot be detected by most of the memory safety approaches, either. We leave this issue to future work.

7.2.4 Mixture of Multiple Pointers

Since AHEMS associates the base and bound information for each pointer (which is 4 bytes long in 32-bit architecture) instead of each byte, an at-

[1] `system("command")` is a `libc` function that executes the `command` in shell.



TID	OID
1	5
2	9
...	...

MEM	OID
0x50	5
0x90	9
...	...

OID	Base	Bound
5	0x12345000	0x1000
9	0x1234A000	0x1000
...	...	...

Figure 7.5: An Illustrative Example of Mixing Two Pointers

tacker may be able to craft a pointer that is a mixture of multiple pointers. Figure 7.5 illustrates a pointer crafted by an attacker using a mixture of two pointers, trying to make a pointer point to an object that is not the intended referent. In Figure 7.5, let's assume that the register `%g1` is supposed to contain the pointer that points to the object inhabiting within `(0x12345000, 0x12346000)`. We can see that the attacker manages to make `%g1` point to another object that resides in `(0x1234A000, 0x1234B000)` by loading the upper half of the pointer at `0x50` to the upper half of `%g1` and loading the lower half of the pointer at `0x90` to the lower half of `%g1`. In this way, the attacker creates a pointer that is a mixture of pointers at `0x50` and `0x90`. The crafted pointer is associated with the base and bound information of the pointer loaded by the last `load` instruction, which is the pointer at `0x90`. However, `%g1` is not supposed to be associated with the base and bound information of the pointer at `0x90`, which leads to a potential problem. We argue that creating a mixture of multiple pointers is very difficult and does not help the attacker much for the following reasons. First of all, the attacker does not have control over the executed code to intentionally perform this kind of operation. Second, loading each part of different pointers to a single register is very uncommon in real-world code because that kind of operation is rarely required in C. Third, even if the attacker can craft a pointer, the crafted pointer still need to stay within the base and bound of the pointer loaded by the last `load` instruction; this guarantees the crafted pointer still accesses a valid object, although maybe not its intended object. Therefore, for an attack like this to succeed, the attacker needs to find the code that has the above-mentioned rare operation and the code must happen to have a vulnerability even when all pointers point to valid objects. We believe the chance of launching an attack like this is very low if not nonexistent. Besides, this limitation is not specific to our approach; all approaches that keep one metadatum for a 4-byte or 8-byte pointer have the same limitation. One potential solution is to use byte-granularity metadata (i.e., associate each byte with a metadatum). Then only if all bytes in a pointer point to the same valid object can the pointer be dereferenced. This guarantees that no use of mixed pointers is allowed. We leave it as future work to add support for byte-granularity metadata.

CHAPTER 8

RELATED WORK

In this chapter, we discuss software and hardware approaches previously proposed to enforce either spatial memory safety, temporal memory safety, or both. Challenges encountered by previous approaches are (1) incomplete memory safety, (2) need for manual annotation or modification to source code, (3) inability of the protected modules to interoperate with unprotected modules, (4) lack of modularity support, (5) prohibitively high runtime overhead, and (6) no physical metadata isolation. Table 8.1 summarizes the comparison of representative approaches with the AHEMS.

8.1 Fat-Pointer Approaches

Some approaches, such as SafeC [74], Cyclone [76, 159], CCured [75], and Fail-Safe C [86], introduce the concept of a *fat pointer* for bound checking. Compared with a regular pointer, a fat pointer contains not only the address of its intended object but also the base and bound. That is, a fat pointer is a 3-tuple (address, base, bound). The base and bound are propagated along with pointer arithmetics and are checked against the address when the pointer is dereferenced. This is an efficient data structure that is used by type-safe languages, such as Java, to perform bound checking. However, since the representation of a pointer in the memory is changed, fat-pointer approaches are not binary-compatible with precompiled binaries, which significantly impedes the deployment of these approaches on existing programs.

CCured [75] performs a type inference algorithm on the whole program to identify pointers that do not need bound checking. It categorizes each pointer as either `SAFE`, `SEQ`, or `WILD`, where a `SAFE` pointer does not need bound checking because it is proven to be in bounds by the algorithm. CCured then inserts runtime bound checks for `SEQ` and `WILD` pointers to enforce spatial memory safety. It ignores explicit deallocation and employs a dynamic garbage collector to enforce temporal memory safety. Although CCured enforces complete memory safety with an average of 28% overhead on Olden benchmarks[1], CCured requires non-trivial manual annotation and modification to the source code to reduce the number of inefficient `WILD` pointers and fix the compatibility issue caused by fat pointers.

Cyclone [76, 159] is a safe dialect of C similar to CCured. It performs static analysis on source code and inserts runtime checks at places where a pointer cannot be statically determined to be safe. Cyclone also requires a programmer to modify the source code, which makes it hard to deploy on large existing programs. It also suffers from fat-pointer problems, even though it reports a moderate runtime overhead (an average of 38% runtime overhead on their 12 tested programs).

Fail-Safe C [86] is a memory-safe compiler of the full ANSI C language. It employs the concept of *fat pointers*, *fat integers*, *typed memory blocks*, *virtual offsets* and *safe memory management* (i.e., garbage collection) to ensure full memory safety that supports casts, unions, arbitrary pointer arithmetics, function pointers, and variable-number arguments. However, it incurs excessively high runtime overhead (an average of 364% on Olden benchmarks) and is not binary compatible and not source compatible due to fat-pointer design.

8.2 Object-based Approaches

Due to the backward compatibility issue of fat-pointer approaches, object-based approaches associate bound information with an object so that the memory layout of each pointer does not need to be changed. This makes the protected code backward compatible with unprotected code, such as the dynamically-linked libraries.

Purify [77] is a commercial tool that detects memory leaks and memory errors. It instruments a program at the object code level at link time, so it does not require any source code changes and has good binary compatibility. Purify conceptually associates a 2-bit status for each byte of the memory to

[1] Olden benchmarks are 10 medium-size programs that are pointer-intensive and used by many approaches for performance evaluation.

track whether the byte is (1) unallocated, (2) allocated but uninitialized, or (3) allocated and initialized. Purify inserts runtime checks at each memory load and store to ensure every store writes to allocated memory and every load reads from allocated and initialized memory. However, Purify fails to detect an out-of-bounds pointer that points to a valid object that is not its intended referent. Also, Purify has very limited protection on stack objects. More importantly, Purify is more of a debugging tool, since it incurs significant runtime overhead (148.44x slowdown on Olden benchmarks).

Jones and Kelly (J&K) [78] propose an object-based technique that maintains an object table, implemented as a splay tree, to keep track of all live objects at any point of execution. When a pointer is dereferenced, the value of the pointer is used to look up the intended referent to retrieve the base and bound information for bound checking. To ensure that a pointer does not point to a valid object that is not the intended referent of the pointer, the approach also needs to make sure the pointer does not go out of bound when pointer arithmetics is performed. This is where object-base approaches get complicated. Because ANSI C Standard allows a pointer to point to the very next byte after an array, the J&K approach needs to pad each object in the program (except for function parameters) to prevent false alarms. The value of a pointer is substituted as an `ILLEGAL` value whenever the pointer goes out of the bound (i.e., goes after the padded byte). However, many existing programs sometimes use an out-of-bounds pointer to compute an in-bounds address, which makes the J&K approach inapplicable to those programs.

CRED [80], building on top of the J&K approach, introduces the OOB object (Out-Of-Bounds object) to solve this problem. Instead of assigning an `ILLEGAL` value to an out-of-bounds pointer, CRED associates each out-of-bounds pointer with an OOB object that contains the information of the pointer's intended referent. When pointer arithmetics is performed on an out-of-bounds pointer, the result is recorded in the OOB object, so that if an out-of-bounds pointer goes back in bounds again, it can be set as an in-bounds pointer without raising false alarms. Although the J&K approach and CRED solve the binary compatibility issue, they still suffer from significant runtime overhead (5-6x slowdown for J&K and 11-12x slowdown for CRED on Olden benchmarks). This is because the object lookup is an expensive operation that searches for the object whose address range contains the value of a given pointer.

Mudflap [79] is a GCC extension that implements the J&K approach, inheriting its pros and cons. The additional benefits of Mudflap are that (1) it uses a lookup cache to improve the object lookup operations, and (2) it uses some heuristics to enhance its binary compatibility with precompiled libraries. However, Mudflap has an average of 175.2x slowdown on Olden benchmarks as measured in our experiment.

SAFECode [81,82], adopting the approach of J&K and CRED, greatly reduces runtime overhead (an average of 12% overhead on Olden benchmarks) by (1) employing the Automatic Pool Allocation to speed up object lookup operations and (2) introducing a level of indirection for the OOB objects to eliminate the need for runtime checks on all load/store instructions. Nonetheless, SAFECode requires inter-procedural, context-sensitive pointer analysis to effectively reduce runtime overhead, which constrains scalability. Since SAFECode needs to modify function interfaces and function calls to include the *pool descriptor* as a parameter, external code that calls internal functions may allow memory safety violations to occur. The reason is that the external code (e.g., precompiled library), which is unavailable to the SAFECode compiler, can only call the unprotected versions of the internal functions due to binary incompatibility. This reduces the detection coverage of memory safety in SAFECode.

Baggy Bound Checking (BBC) [83,160], which also builds on top of J&K, further reduces the runtime overhead of the object lookup operations by allocating memory using *buddy memory allocation* and enforcing pointers to stay within the *allocation bounds* instead of the actual bounds of an object. Specifically, BBC only allocates memory regions of sizes that are powers of 2, so that the allocation size can be represented by only one byte. Because BBC makes each allocated memory region align to its size, the base address of an object can be computed by clearing the $\log_2(s)$ least significant bits of a pointer, where s is the size of the object pointed by the pointer. Since the base address of an object can be computed, the object table only needs one byte for each object to store the bound. With the above setup, the object table can be just a contiguous array, which makes the object lookup very efficient. BBC even removes all bound checks on pointer dereferences and only perform checks on pointer arithmetics to optimize performance. Although BBC achieves low runtime overhead (an average of 6% overhead on Olden benchmarks), it trades the completeness of memory safety for performance because it allows more temporal memory errors than J&K due to not checking on pointer dereferences. Further, BBC does not interoperate with precompiled libraries that may call the functions it protect because BBC changes the stack frame layout.

AddressSanitizer [84] represents another class of object-based approaches that has weaker guarantee on spatial memory safety than J&K. These approaches use a larger padding (called *redzones*) to reduce the possibility of a pointer going out of bounds to reach another valid object that is not the intended referent of the pointer. With the protection of large paddings at the beginning and the end of each object, all runtime checking on pointer arithmetics is removed and only pointer dereferences are checked. AddressSanitizer adopts this technique and dedicates one eighth of the virtual memory as the object table for efficient object lookup. Similar to BBC, AddressSanitizer makes all allocated objects align to 8 bytes. Each 8-byte region has an entry, which is only one byte, in the object table to represent how many bytes in this 8-byte region are valid. With this setup, the AddressSanitizer can look up the object table with only one binary shift and one addition. Although AddressSanitizer has reasonable performance overhead (144% on Olden benchmarks in our experiments[2]), it has the same binary compatibility problem as BBC does, since it also changes the stack frame layout.

In contrast to AddressSanitizer, PAriCheck [85] inserts runtime checks only on pointer arithmetics instead of only on pointer dereferences. PAriCheck adopts the approach of CRED but improves the speed of object lookup by storing a unique ID instead of bounds in the object table and aligning all allocated objects to 2^k bytes, where k is configurable so that the size of the object table becomes small enough to fit in a contiguous array. During pointer arithmetics, one only needs to check whether the ID associated with the original location pointed by the pointer matches the ID of the resultant location. This gives PAriCheck an even lower performance overhead than BBC (average 4.5% overhead on Olden). However, without the runtime checking on pointer dereferences, PAriCheck has the same problem as BBC: smaller coverage on temporal memory errors (i.e., use-after-free or use of dangling pointers).

In general, object-based approaches can detect most of spatial memory

[2] The runtime overhead measured in our experiments is generally higher than that reported by the authors due to the reasons discussed in Chapter 6.

errors except for sub-object overflows. However, they only provide limited protection for temporal memory errors. For example, if the intended referent of a pointer `p` is freed and subsequent allocation reuses the same memory region as the freed object to create an unrelated object, the access to the unrelated object using the pointer `p` causes a temporal memory error that evades the detection of object-based approaches.

8.3 Pointer-based Approaches

In contrast to object-based approaches, pointer-based approaches associate base and bounds metadata with the pointer instead of the object. Metadata are propagated along with pointer arithmetics, and runtime checks are performed only on pointer dereferences, since an out-of-bounds pointer does not harm anything as long as it is not dereferenced. The benefits of pointer-based approaches are that (1) detection of sub-object overflows becomes possible because a pointer can be associated with only the sub-base and sub-bound of the sub-object, (2) no padding needs to be added to the end of an object because there is no need to determine which object a pointer points to during pointer arithmetics, (3) the cost of object table lookup is generally lower than that of object-based approaches, since an object table can be implemented as a hash table or a contiguous array, and (4) it is possible to ensure both spatial and temporal memory safety without combining with other methods. However, pointer-base approaches are not perfect. The binary compatibility of software pointer-based approaches is still limited. Since pointer-based approaches need to propagate the base and bound metadata along with the pointer arithmetics but the propagation of metadata is not performed in the unprotected code, the metadata of a pointer may become outdated after a function call to unprotected code. For example, if a pointer is passed as an argument from protected code to unprotected code and the pointer is modified in the unprotected code, the metadata of the pointer are outdated and subsequent dereferences of the pointer may allow memory errors to occur. In contrast, AHEMS has much better binary compatibility in that the metadata can be updated even in the unprotected code since propagation is handled by the hardware. Furthermore, AHEMS can perform the bound checks on pointer dereferences of those pointers generated by protected code and used in the unprotected code. Now we will discuss some representative pointer-based approaches and their limitations.

SafeC [74] is a pointer-based approach that uses fat-pointer representation to associate the metadata of base, bound, storage class, and capability with the pointer to enforce complete memory safety. However, it is not binary compatible with precompiled libraries, since programmers or systems may make assumptions about the size of a pointer. SafeC is not source compatible because it requires users to manually translate the declaration of pointers, use of pointers, and calls to `malloc()` and `free()` to their MACROs, which impedes deployment. Furthermore, the runtime overhead of SafeC is relatively high (130%-540% on Olden benchmarks).

The Patil and Fischer [88] approach, closely related to the approach of SafeC, automates the insertion of runtime checks and separates the metadata from the pointer to achieve better source compatibility and binary compatibility, respectively. Also, they improve the performance by (1) faster temporal safety checks with the *lock address* and *key* technique, (2) customization, and (3) shadow processing. For temporal safety, each pointer is associated with a pair of *lock address* and *key*. The pair of *lock address* and *key* is propagated along with pointer arithmetics. When a pointer is created, the value stored in the *lock address* is set to the value of *key*. When the object pointed to by the pointer is deallocated, the value in the *lock address* is set as invalid. All pointers pointing to the same object have the same *lock address* and *key*. Therefore, the object pointed to by a pointer is live as long as the value in the *lock address* matches the *key* associated with the pointer. The customization throws away computations not relevant to runtime checking, while the shadow processing offloads the runtime checking to another idle processor to be performed in the background. Although customization and shadow processing make original programs that offload the runtime checking run with less than 10% overhead, their customized shadow processes that perform the runtime checking for the original programs still have an average of 518% slowdown, as reported in their experiments, which is even higher than that of SafeC. Furthermore, the backward-compatibility of the Patil and Fischer approach is limited because their approach needs to change the interfaces of functions, which causes the same problem as in SAFECode.

Memory Safety C Compiler (MSCC) [87] also uses the technique conceptually similar to that of Patil and Fischer except for shadow processing (i.e., MSCC does not offload runtime checks to idle processors). MSCC adds another security feature to prevent unsafe downcasts by maintaining RTTI (Runtime Type Information) with each object. MSCC further improves the performance overhead of SafeC and the Patil and Fischer approach, reducing the runtime overhead to an average of 99–133% on Olden benchmarks. However, MSCC's optimization makes it lose the ability to detect sub-object overflows.

SoftBound [89] and CETS (Compiler-Enforced Temporal Safety) [90] are also pointer-based approaches that protect spatial memory safety and temporal memory safety, respectively. They adopt an idea similar to that of Patil and Fischer but further improve both detection coverage and performance overhead. SoftBound can detect sub-object overflows that may be missed by MSCC and allows arbitrary casts between pointers that are not handled by MSCC or Patil and Fischer. SoftBound + CETS together has been proven to guarantee complete memory safety. In terms of performance, SoftBound + CETS achieves an average of 116% runtime overhead on 17 benchmarks selected from SPECINT and SPECFP[3], as reported by the authors, and 380% overhead on Olden benchmarks in our experiment. Although SoftBound + CETS enforce complete memory safety with moderate performance overhead, they still suffer from the binary compatibility issue of all software approaches, as discussed above.

MemSafe [91, 92] models temporal errors as spatial errors so that one runtime check for each pointer dereference is enough for both spatial and memory safety. In particular, MemSafe transforms a program on memory deallocations and pointer stores so that each pointer assignment is a direct assignment[4]. For the memory deallocation, each `free(*ptr)` function call is appended by `ptr = INVALID` so that memory deallocation is treated as a pointer assignment. This essentially transforms temporal errors into spatial errors. For the pointer stores, MemSafe inserts a ϱ-function (like a ϕ-function in SSA form) after each pointer load to indicate all possible direct objects that can be pointed by the pointer. For example, if `q` is of type `int**` that can possibly point to `a` or `b` of type `int*`, then MemSafe will insert

[3] SPEC benchmarks [161] are large-size programs that are standards for performance evaluation

[4] A direct assignment is an assignment whose right hand side is directly the object being assigned (i.e., not a pointer dereference).

$\texttt{c} = \varrho(\texttt{a}, \texttt{b})$ after the assignment `c = *q`. The above two transformations allow MemSafe to compute a Data Flow for Pointers Graph (DFPG) that can speed up the runtime checking of MemSafe. Although MemSafe enforces both spatial and memory safety while incurring low runtime overhead (an average of 87% overhead on Olden, PtrDist [74], and SPEC benchmarks), it requires whole-program analysis to perform the transformations, which limits its modularity support and scalability.

8.4 Other Software-only Approaches

Yong and Horwitz [152] proposes a technique to ensure each write via an unsafe pointer always writes to *appropriate* targets of the write. They first perform static analysis on the source code to identify unsafe pointers and their points-to sets. Each byte of allocated memory is associated with a bit to indicate whether the byte is appropriate for some unsafe pointer. When a write via an unsafe pointer occurs, the associated bits of the memory location being written for all *appropriate* targets of the unsafe pointer are checked to see if the write is allowed. They also do the runtime checking on each call to `free()` to ensure a limited temporal memory safety. However, since their approach requires static analysis, modularity support and scalability are limited. Their binary compatibility is also limited because they may generate false positives when interoperating with unprotected code. Also, the points-to sets computed statically are conservative, so an attacker can still have a limited ability to write as long as the attack target happens to be in the same points-to set of the manipulated write. More importantly, they do not check all reads, which may allow information leakage attacks. Their average runtime overhead on Olden benchmarks is 37%.

WIT [46] is an approach similar to that of Yong and Horwitz. WIT improves the detection coverage by using more than just one bit to reduce the possibility of an attacker writing to an object that is not the intended referent of the write but happens to be in the same points-to set of the intended referent. WIT also inserts canaries after each allocated object to detect buffer overflows, and it enforces control-flow integrity to compensate the imprecise points-to sets. With all the above checks added, WIT still has lower performance overhead (average 4% overhead on Olden) than that of

Yong and Horwitz. However, it inherits some deficiencies from the approach of Yong and Horwitz, including the limited modularity support, scalability, and binary compatibility.

SymPLAID [162] is a formal technique that utilizes symbolic execution to statically identify all variables that are vulnerable to insider attacks (a superset of memory corruption attacks) at each attack point (the specific point in the execution of a program where such an attack may happen). For each attack point, SymPLAID assigns a symbolic value to one of the variables and then continue execution by treating the symbolic value as any possible value. Thus the execution may fork during branch instructions, since both paths are possible. The symbolic value can be constrained when compared to some known value. For example, if `sym_val < 10` is encountered and `sym_val` is a symbolic value, then the execution forks into two executions with one having the constraint of `sym_val < 10` and the other having the constraint of `sym_val >= 10`. By checking whether the result of the execution is as expected, SymPLAID can identify those variables that are vulnerable to insider attacks. Although SymPLAID can find memory errors statically, it suffers from a scalability issue: it essentially enumerates all possible states and the number of explored states grow exponentially with the program's size and address space.

8.5 Hardware Approaches

SafeMem [99] utilizes the ECC bits existing in memory to detect memory leaks and some classes of memory errors. It proposes an approach similar to page protection but with finer granularity (cache-line granularity instead of page granularity), which reduces the amount of false sharing and padding space. Specifically, SafeMem temporarily disables the ECC protection and flips 3 bits on the data to make ECC code invalid. This enables the operating system to be notified whenever the data is accessed, since the ECC code does not match the one computed from the new data. Although ECC code is checked on memory reads but not on memory writes, the use of cache in modern systems enables SafeMem to read the memory before every write by flushing the cache line of the protected zone to the memory. This is because in this case every write to the protected zone must first load the memory to the cache line, which triggers the ECC check. SafeMem detects buffer overflows by padding redzones around a buffer and uses ECC to catch the access of redzones caused by buffer overflow. SafeMem also detects invalid access to freed memory by using the ECC protection to watch all freed memory. An invalid access to freed memory will invoke the ECC fault handler. The advantages of SafeMem is that it does not need additional hardware to be installed and it can achieve low runtime overhead (an average of 6.3% overhead on 3 non-standard benchmarks). However, due to the limitations of ECC, SafeMem fails to prevent some classes of attacks, such as the attacks that allow arbitrary writes. Also, if the data being modified by an attacker is not yet written back to the memory (i.e., the data stays in the cache), SafeMem may miss or delay the detection of the attack. Such an attack would be the non-control data attacks launched by overflowing a decision-making data in a program stack.

MemTracker [100, 101] is a hardware-programmable state machine residing in memory that associates each byte of memory with a *state* and treats each access to the memory as an *event*. When an *event* is triggered, the state machine can transition to the next state according to a Programmable State Transition Table (PSTT). MemTracker implements three example protections to protect (1) allocation/deallocation of heap memory, (2) return address, and (3) heap buffer overflow. Although MemTracker achieves as low as 2.7% runtime overhead on SPEC benchmark, it only detects the abovementioned three classes of memory errors and fails to detect other memory corruption attacks.

Clause et al. [95,96] associate each pointer and each byte of memory with a *taint mark* and require that only pointers having the same taint mark as the memory can access the memory. When a region of memory is allocated, each byte of the memory is tainted with a unique taint mark. When a pointer is initialized to point to some memory, the pointer is tainted with the same taint mark as the memory. During pointer arithmetics, taint marks of pointers are propagated. When a pointer is dereferenced, the taint mark of the pointer is checked to see if it matches that of the pointed byte of the memory. Clause et al. implement the taint propagation rules and taint checks in hardware to reduce runtime overheads. They achieve an average of 13% runtime overhead on SPEC2000 benchmarks when 256 different taint marks are used. However, the approach of Clause et al. may have false negatives because the number of taint marks is limited. Besides, they may also incur significant runtime overhead if large memory allocation and deallocation are very frequent, as their approach needs to set a unique taint mark for each byte of allocated or deallocated memory.

Arora et al. purpose an architectural support on embedded processors to reduce the performance overhead of CCured. They extend the ISA with `XCHECK` instructions that allow the processor to efficiently perform the runtime checking on `SAFE`, `SEQ`, and `FSEQ`, and `WILD` pointers. Arora et al. implement their approach on a cycle-accurate simulator of the Xtensa [163] processor that enables them to evaluate the hardware overhead, which is less than 10%. However, although they improve the performance overhead of CCured by an average of 2.3x on Olden benchmark, their approach still inherits the deficiencies of CCured, as discussed above.

SafeProc [97] extends the ISA of a processor with safety instructions and adds architectural support to the processor to help detect memory errors. SafeProc conceptually employs the pointer-based approach by maintaining a table in hardware to keep track of the base and bound information for each pointer. The lookup table is implemented as three CAMs (Content-Addressable Memories) on chip and a BRS (Backup Record Storage) that resides in off-chip memory and stores the entries evicted by CAMs. The search for a pointer is hierarchical: if the base and bound information of a pointer cannot be found in the CAMs, the BRS is searched. Memory deallocation becomes an expensive operation because SafeProc needs to search all entries to delete all entries associated with the deallocated memory. SafeProc also proposes an optimization to delay memory error detection whenever possible, which is similar to the asynchronous property of AHEMS. However, the degree of asynchrony in SafeProc is very limited compared with AHEMS because SafeProc cannot delay the detection if dependency between instructions exists, which is not uncommon. For example, if `%g1` is the destination register of one instruction as well as the source register of the next instruction, the memory errors detection cannot be delayed. Otherwise, the base and bound of `%g1` may not be properly propagated. Also, SafeProc requires manual modification of a program to insert safety instructions, which is an obstacle for deploying SafeProc on existing programs. The average runtime overhead of SafeProc is 9% on Olden benchmarks.

Chuang et al. [98] accelerate the metadata lookup of the pointer-based approach similar to SafeC to improve performance overhead. They store metadata inline with objects or pointers just like a fat pointer or a fat object so that hardware does not need to take care of metadata propagation during pointer arithmetics (i.e., the compiler takes care of metadata propagation). Chuang et al. introduce a *meta-check* instruction in the ISA to bind a check operation to a virtual register. The *meta-check* instruction notifies the hardware what type of check should be performed when the virtual register is dereferenced. They evaluate the performance overhead for both the case of associating metadata with objects and associating metadata with pointers. They conclude that associating metadata with objects is better in terms of performance. Chuang et al. also evaluate the performance when the metadata is compressed so that there are fewer loads or stores needed for the metadata. In their experiment, they achieve around 21.2% runtime overhead on SPECINT benchmark. However, the use of fat pointers or fat objects makes the approach of Chuang et al. not binary-compatible since the memory layout is changed.

HardBound [93] is essentially a hardware implementation version of SoftBound that enforces spatial memory safety. In contrast to Chuang et al., HardBound separates the metadata from the pointer when a pointer is stored into the memory. HardBound widens each register in the processor with a base and a bound so that the base and bound are propagated along with pointer arithmetics in hardware without requiring additional cycle. When a pointer is created, HardBound instruments a `setbound` instruction in the source code to set up the base and bound for the pointer. When a pointer is dereferenced, HardBound checks whether the pointer is within the base and bound to check spatial memory safety. When a pointer is stored into or loaded from the memory, its base and bound need to be stored into or loaded from the shadow memory. Therefore, HardBound can be optimized by encoding the base and bound information in the 4 MSBs of a pointer to reduce the number of loads and stores needed for base and bound in common cases. HardBound incurs an average of 7-9% overhead on Olden benchmarks.

Watchdog [94], which is the state-of-the-art hardware approach, improves HardBound by implementing a hardware version of CETS that ensures temporal memory safety, so as to achieve full memory safety when combining with HardBound. In addition to the base and bound, Watchdog further associates each register with a pair of *lock address* and *key*. Watchdog inserts μops for each `load`, `store`, `call`, `return`, `malloc`, `free`, and pointer arithmetics operation in the processor to maintain and check the validity of *lock address* and *key*. Similar to HardBound, Watchdog also has a shadow memory to contain those metadata when a pointer is stored into or loaded from the memory. Watchdog incurs only 24% performance overhead on SPEC benchmarks for complete memory safety. Compared with Watchdog, AHEMS achieves the same level of complete memory safety and runtime overhead as Watchdog while having additional advantages over Watchdog. First, AHEMS is more practical because it only needs to widen the size of each register by $\log_2 n$ bits where n is the number of registers in a processor, while Watchdog needs to quintuple the size of each register to keep 4 additional metadata *base*, *bound*, *lock address* and *key*. This means that Watchdog needs to add an additional 256 bits to each register in a 64-bit architecture. This is not only a prohibitively high hardware resource overhead to the on-chip memory, but it is lethal to the critical path of the processor. Second, we have a real FPGA implementation of AHEMS that shows that AHEMS is a realistic design and allows us to measure more accurate performance, hardware resource, and power consumption overhead of our prototype than an instruction-accurate simulator used by Watchdog. Our experiment also shows that AHEMS has very little impact on the critical path of the main processor. Third, AHEMS provides physical isolation of the metadata that prevents the attackers from modifying the metadata by exploiting other vulnerabilities of the system.

Table 8.1: Comparison of Representative Memory Safety Approaches

Approach	**Spatial memory safety**	**Temporal memory safety**	**Source compat.**	**Binary compat.**	**Modular support**	**Scalability**	**Physical meta-data isolation**	**Slowdown**[a]
Fat-pointer Approaches								
SafeC [74]	Yes	Yes	No	No	Yes	Yes	No	2.3x-6.4x
CCured [75]	Yes	Yes	No	No	No	No	No	1.28x
Cyclone [76]	Yes	Yes	No	No	No	No	No	1.38x (non-standard)
Object-based Approaches								
Purify [77]	Yes	Incomplete	Yes	Yes	Yes	Yes	No	148.44x
J & K [78]	Yes	Incomplete	Yes	Yes	Yes	Yes	No	5x-6x
Mudflap [79]	Yes	Incomplete	Yes	Yes	Yes	Yes	No	175.20x
CRED [80]	Yes	Incomplete	Yes	Yes	Yes	Yes	No	11x-12x
SAFECode [81, 82]	Yes	Incomplete	Yes	Yes	Yes	Limited	No	1.12x[b]
BBC [83, 160]	Yes	Incomplete	Yes	Limited	Yes	Yes	No	1.06x
Address Sanitizer [84]	Yes	Incomplete	Yes	Limited	Yes	Yes	No	2.44x
PAriCheck [85]	Yes	Incomplete	Yes	Limited	Yes	Yes	No	1.05x
Fail-Safe C [86]	Yes	Yes	No	No	Yes	Yes	No	4.64x
Pointer-based Approaches								
MSCC [87]	Yes	Yes	Yes	Limited	Yes	Yes	No	1.99x-2.33x
Patil & Fischer [88]	Yes	Yes	Yes	Limited	No	Yes	No	6.18x (non-standard)
SoftBound [89]	Yes	No	Yes	Limited	Yes	Yes	No	1.67x-1.93x (Olden+SPEC)
CETS [90]	No	Yes	Yes	Limited	Yes	Yes	No	1.48x (SPEC)
SoftBound + CETS	Yes	Yes	Yes	Limited	Yes	Yes	No	4.8x
MemSafe [91, 92]	Yes	Yes	Yes	Limited	Limited	Limited	No	1.29x
Other Software-only Approaches								
Yong and Horwitz [152]	Only writes	Limited	Yes	Limited	Limited	Limited	No	1.37x
WIT [46]	Only writes	Limited	Yes	Limited	Limited	Limited	No	1.04x
Hardware Approaches								
SafeMem [99]	Incomplete	Incomplete	Yes	Yes	Yes	Yes	No	1.06x (non-standard)
MemTracker [100, 101]	Incomplete	Incomplete	Yes	Yes	Yes	Yes	No	1.03x (SPEC)
Clause et al. [95, 96]	Incomplete	Incomplete	Yes	Yes	Yes	Yes	No	1.05x-1.21x (SPEC)
Arora et al. [150]	Yes	Yes	No	No	No	No	No	1.5x (non-standard)
SafeProc [97]	Yes	Yes	No	Limited	Yes	Yes	No	1.09x
Chuang et al. [98]	Yes	Yes	Yes	No	Yes	Yes	No	1.21x (SPEC)
HardBound [93]	Yes	No	Yes	Yes	Yes	Yes	No	1.07x-1.09x
Watchdog [94]	Yes	Yes	Yes	Yes	Yes	Yes	No	1.24x (SPEC)
AHEMS	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	1.11x

[a] The runtime overhead is reported by the authors for Olden benchmarks unless otherwise specified.

[b] Note that the runtime overhead of SAFECode is different from the overhead measured in our experiments due to the reasons explained in Section 6.3.2.

CHAPTER 9

CONCLUSION AND FUTURE WORK

In this chapter, we draw a conclusion for this thesis and discuss some possible directions for the future work.

9.1 Conclusion

This thesis presents AHEMS, an architectural support for *asynchronous checking* to ensure both spatial and temporal memory safety. AHEMS is a pointer-based approach that associates each pointer with a base and bound metadata, propagates the metadata during pointer arithmetics, and checks the bounds and lifetime of the accessed object during pointer dereference. *Asynchronous checking* is accomplished by the coordination of a *runtime monitor* (embedded in the *main processor*) and a *security engine* that can be deployed inside the processor, as a co-processor, or as an external device. The *runtime monitor synchronously* keeps track of each memory event (allocation/deallocation/load/store), maintains the validity of the `TID` (an in-processor metadatum) of each register, and notifies the *security engine* of each memory event. The *security engine asynchronously* processes each received event to update the `TID2OID`, `MEM2OID` and `OID2BaseBound` tables and to check the base and bound of each memory load and store using these three tables. AHEMS enables system designers to flexibly decide where to place the *security engine*, depending on system requirements. *Asynchronous checking* greatly reduces the runtime overhead caused by memory safety checking, since checks can be performed asynchronously and with different frequency from the *main processor*. Finally, the metadata are stored in a different physical memory to guarantee the integrity of the metadata. To sum up, AHEMS enforces spatial and temporal memory safety with low runtime overhead (10.6% overhead on Olden benchmarks), negligible impact on critical path (0.06% overhead) and power consumption (0.5% overhead), physical metadata isolation, high flexibility, high scalability, and support for source compatibility, binary compatibility, and modularity.

9.2 Future Work

In the following, we discuss some future directions that continue or extend the idea of AHEMS, along with some possible improvements to our implementation prototype.

9.2.1 Non-pointer Load/Store Instructions

One way to improve the runtime overhead of AHEMS is to reduce the number of load and store events sent to the security engine. If the ISA of the main processor can be extended by introducing new `opcode`s to indicate whether a `load/store` instruction accesses a pointer, AHEMS can skip the steps that maintain the `MEM2OID` table for those non-pointer `load/store` instructions because the `MEM2OID` table is only used to keep track of in-memory pointers. One can profile the execution of a program and modify the compiler to mark those `load/store` instructions that do not access pointers as non-pointer ones using the extended `opcode`s, so that, in runtime, the security engine can skip some steps for those loads/stores.

9.2.2 Checking for Only Critical Functions

If AHEMS can be extended to protect only critical functions and skip memory safety checking on trusted functions, the runtime overhead can be even lowered while maintaining the memory safety of those critical functions. AHEMS can then fully utilize the execution time of trusted functions to perform the checking, due to its asynchronous property. This can potentially be achieved by introducing two machine instructions, `stop_checking` and `start_checking`, and instrumenting the source code with a `stop_checking` before each trusted function call and with a `start_checking` after each unprotected function call. The same idea can be applied on trusted kernel functions and context switches.

9.2.3 Checking for Type Safety

To prevent an object of one type from being treated as an object of another type, AHEMS needs to be enhanced to ensure type safety. One potential solution is to associate not only the base and bound information but also the type information with each pointer. Also needed are checks in pointer arithmetics as to whether the type of the source pointer can be upcast[1] to the type of the destination pointer. A new instruction needs to be introduced to explicitly perform type casting so that the type associated with an pointer can be changed according to the source code. The source code should also be instrumented with the new instruction to notify the runtime monitor about each type casting. The challenges here are how to maintain the source compatibility and binary compatibility of AHEMS, since many programs are written in ways that violate type safety.

9.2.4 Cache for the Security Engine

In our implementation prototype, we do not use a cache to speed up the memory access of metadata. If a cache can be added between the security engine and the dedicated memory for metadata, the performance of safety checking can definitely be sped up. The `MEM2OID` table has similar locality as the memory footprints of the executed programs and thus can benefit from the cache.

9.2.5 Multiprocess Support

At this stage of development, our implementation prototype only enforces memory safety for one process at a time. The process can be multithreaded because the prototype recognizes the process by the specific identifier used by an MMU to differentiate process address spaces. To support memory safety checking for multiple processes, an MMU can be added to the security engine to put `MEM2OID` tables of different processes into different address spaces. During context switches, the security engine needs to switch from

[1] One type `T1` can be upcast to another type `T2` if some prefix structure of `T1` contains the structure of `T2`. For example, `struct Audi {char *type; float price; char *model}` can be upcast to `struct Car {char *type; float price;}`.

the old MEM2OID table to the new MEM2OID table and reload TID2OID table for the to-be-executed process. All processes can share one OID2BaseBound table since OID can be unique among all processes.

9.2.6 In-processor Design and External-device Design

In this thesis, we describe three possible placements of the security engine, each of which has its advantages and disadvantages. We have compared those three designs qualitatively and evaluated the co-processor design using our implementation prototype in Chapter 6 and Chapter 7. It would be beneficial to also implement prototypes of the in-processor design and the external-device design to compare their pros and cons quantitatively, which will help system designers make better decisions when deploying AHEMS on their systems. For example, one can examine the runtime overhead and hardware overhead of the in-processor design on an out-of-order processor, or measure the communication overhead on PCI-E caused by the external-device design.

REFERENCES

[1] TIOBE Software BV, “TIOBE programming community index,” 2013. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] DedaSys LLC, “Programming language popularity,” 2011. [Online]. Available: http://langpop.com/

[3] James P. Anderson, “Computer security technology planning study,” Deputy for Command and Management Systems, Tech. Rep. ESD-TR-73-51, Oct. 1972.

[4] “CWE/SANS top 25 most dangerous software errors,” 2011. [Online]. Available: http://cwe.mitre.org/top25/

[5] The PaX Team, “The original design & implementation of PAGEEXEC,” Tech. Rep., 2000.

[6] The PaX Team, “paging based non-executable pages,” Tech. Rep., 2003.

[7] Arjan van de Ven, “New security enhancements in red hat enterprise linux v.3, update 3,” Red Hat Inc., Tech. Rep., 2004. [Online]. Available: http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

[8] Microsoft, “A detailed description of the data execution prevention (DEP) feature in windows XP service pack 2, windows XP tablet PC edition 2005, and windows server 2003,” 2006. [Online]. Available: http://support.microsoft.com/kb/875352

[9] Theo de Raadt, “The OpenBSD 3.3 release,” 2003. [Online]. Available: http://www.openbsd.org/33.html

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, “StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th USENIX Security Symposium*, 1998.

[11] Stack Shield, “A stack smashing technique protection tool for linux,” 1999. [Online]. Available: http://www.angelfire.com/sk/stackshield/

[12] Hiroaki Etoh, “GCC extension for protecting applications from stack-smashing attacks,” 2001. [Online]. Available: http://www.research.ibm.com/trl/projects/security/ssp/

[13] Nick Nikiforakis, Frank Piessens, and Wouter Joosen, “HeapSentry: kernel-assisted protection against heap overflows,” in *10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013.

[14] Q. Zeng, D. Wu, and P. Liu, “Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI'11)*, June 2011, p. 367–377.

[15] Y. Younan, W. Joosen, and F. Piessens, “Efficient protection against heap-based buffer overflows without resorting to magic,” in *Proceedings of the 8th international conference on Information and Communications Security (ICICS'06).* Berlin, Heidelberg: Springer-Verlag, 2006, p. 379–398.

[16] Emery D. Berger, “HeapShield: library-based heap overflow protection for free,” University of Massachusetts Amherst, Tech. Rep., 2006.

[17] E. D. Berger and B. G. Zorn, “DieHard: probabilistic memory safety for unsafe languages,” in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, June 2006, pp. 158–168.

[18] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, “Archipelago: trading address space for reliability and security,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'08)*, Mar. 2008, pp. 115–124.

[19] G. Novark and E. D. Berger, “DieHarder: securing the heap,” in *Proceedings of the 17th ACM conference on Computer and communications security (CCS'10).* New York, NY: ACM, 2010, p. 573–584.

[20] J. Xu, Z. Kalbarczyk, and R. Iyer, “Transparent runtime randomization for security,” in *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings*, Oct. 2003, pp. 260 – 269.

[21] G. S. Kc, A. D. Keromytis, and V. Prevelakis, “Countering code-injection attacks with instruction-set randomization,” in *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03).* New York, NY, USA: ACM, 2003, p. 272–280.

[22] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP’12)*, San Francisco, CA, 2012, p. 601–615.

[23] The PaX Team, “Address space layout randomization,” 2001. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[24] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of the 11th ACM conference on Computer and communications security (CCS’04)*, Washington, DC, 2004, p. 298–307.

[25] Payer, Mathias, “Too much PIE is bad for performance,” ETH Zurich, Department of Computer Science, Tech. Rep., 2012.

[26] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, “Binary stirring: self-randomizing instruction addresses of legacy x86 binary code,” in *Proceedings of the 19th ACM conference on Computer and communications security (CCS’12)*, Raleigh, NC, 2012, p. 157–168.

[27] S. Bhatkar and R. Sekar, “Data space randomization,” in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’08).* Berlin, Heidelberg: Springer-Verlag, 2008, p. 1–22.

[28] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, “Breaking the memory secrecy assumption,” in *Proceedings of the Second European Workshop on System Security (EUROSEC’09)*, Nuremberg, Germany, 2009, p. 1–8.

[29] C. Cadar, D. Dunbar, and D. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI’08)*, Berkeley, CA, 2008, p. 209–224.

[30] Yannick Moy, Nikolaj Bjorner, and Dave Sielaff, “Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis,” Microsoft Research, Tech. Rep. MSR-TR-2009-57, 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/?id=80722

[31] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary linux programs,” in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM’09. Berkeley, CA, USA: USENIX Association, 2009, p. 67–82.

[32] Tielei Wang, T. Wei, Zhiqiang Lin, and Wei Zou, "IntScope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, 2009.

[33] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of c and c++ programs using a compiler IR," in *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments (VSTTE'12)*. Berlin, Heidelberg: Springer-Verlag, 2012, p. 146–161.

[34] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, 2012, p. 163–177.

[35] Oded Horovitz, "Big loop integer protection," *Phrack Magazine*, vol. 11, no. 60, 2002. [Online]. Available: http://www.phrack.com/issues.html?issue=60&id=9

[36] Ramkumar Chinchani, Anusha Iyer, Bharat Jayaraman, and Shambhu Upadhyaya, "ARCHERR: runtime environment driven program safety," in *Proceedings of the 9th European Symposium on Research in Computer Security*, Sophia Antipolis, France, 2004, pp. 385–406.

[37] Rafal Wojtczuk, "UQBTng: a tool capable of automatically finding integer overflows in win32 binaries," in *Proceedings of the 22nd Chaos Communication Congress*, Berlin, Germany, 2005.

[38] David Brumley, Tzi-cker Chiueh, Rob Johnson, Huijia Lin, and Dawn Song, "RICH: automatically protecting against integer-based vulnerabilities," in *the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, California, 2007.

[39] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie, "IntFinder: automatically detecting integer bugs in x86 binary program," in *Proceedings of the 11th international conference on Information and Communications Security (ICICS'09)*. Berlin, Heidelberg: Springer-Verlag, 2009, p. 336–345.

[40] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "IntPatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *Proceedings of the 15th European conference on Research in computer security (ESORICS'10)*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 71–86.

[41] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proceedings of the 2012 International Conference on Software Engineering (ICSE'12)*, Piscataway, NJ, 2012, p. 760–770.

[42] Emese Revfy, "Inside the size overflow plugin," 2012. [Online]. Available: https://forums.grsecurity.net/viewtopic.php?f=7&t=3043

[43] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002, p. 191–206.

[44] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security (CCS'05).* New York, NY, USA: ACM, 2005, p. 340–353.

[45] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009.

[46] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (SP'08)*, Oakland, CA, May 2008, pp. 263–277.

[47] Z. Wang and X. Jiang, "HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP'10)*, Oakland, CA, 2010, p. 380–395.

[48] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, Orlando, FL, 2011, p. 353–362.

[49] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter, "Code pointer masking: hardening applications against code injection attacks," in *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment (DIMVA'11).* Berlin, Heidelberg: Springer-Verlag, 2011, p. 194–213.

[50] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: detecting violation of control flow integrity using performance counters," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, June 2012, pp. 1 –12.

[51] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS'12)*, 2012.

[52] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres, "Protecting function pointers in binary," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS'13)*, Hangzhou, China, 2013, p. 487–492.

[53] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou, "Practical control flow integrity & randomization for binary executables," in *Proceedings of 34th IEEE Symposium on Security & Privacy (SP'13)*, San Francisco, CA, 2013.

[54] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS'13)*, Hangzhou, China, 2013, p. 311–322.

[55] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (PLDI'04)*, Oct. 2004, p. 85–96.

[56] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, Dec. 2004, pp. 221– 232.

[57] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August, "RIFLE: an architectural framework for user-centric information-flow security," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*. IEEE, 2004, pp. 243–254.

[58] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*, June 2007.

[59] M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace & kernelspace," in *Proceedings of the 17th Conference on Security Symposium (SS'08)*, Berkeley, CA, 2008, p. 395–410.

[60] James Newsome and Dawn Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, Feb. 2005.

[61] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "LIFT: a low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, p. 135–148.

[62] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[63] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006, p. 749–754.

[64] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA'07)*, 2007, p. 196–206.

[65] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'08)*, Mar. 2008, p. 308–318.

[66] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*, 2008, p. 1–14.

[67] A. Ermolinskiy, S. Katti, S. Shenker, L. L. Fowler, and M. McCauley, "Towards practical taint tracking," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92, Jun 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-92.html

[68] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection (RAID'11)*. Berlin, Heidelberg: Springer-Verlag, 2011, p. 1–20.

[69] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, California, Feb. 2011.

[70] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, p. 142–154, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945039

[71] A. Zavou, G. Portokalidis, and A. D. Keromytis, "Taint-exchange: a generic system for cross-process and cross-host taint tracking," in *Proceedings of the 6th International conference on Advances in information and computer security (IWSEC'11).* Berlin, Heidelberg: Springer-Verlag, 2011, p. 113–128.

[72] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE'12)*, 2012, p. 121–132.

[73] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*, 2006, p. 147–160.

[74] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI'94)*, 1994, p. 290–301.

[75] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, May 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1065892

[76] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATEC'02)*, 2002, p. 275–288.

[77] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the USENIX Winter Technical Conference*, 1992, pp. 125–136.

[78] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors, “Backwards-compatible bounds checking for arrays and pointers in c programs,” in *Third International Workshop on Automated Debugging*, 1997, p. 255–283.

[79] Frank Ch. Eigler, “Mudflap: Pointer use checking for C/C++,” in *GCC Developer’s Summit*, 2003.

[80] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in *In Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS’04)*, 2004, p. 159–169.

[81] D. Dhurjati, S. Kowshik, and V. Adve, “SAFECode: enforcing alias analysis for weakly typed languages,” in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI’06)*, June 2006.

[82] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *Proceedings of the 28th international conference on Software engineering (ICSE’06)*, ser. ICSE ’06, 2006, p. 162–171.

[83] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors,” in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM’09. Berkeley, CA, USA: USENIX Association, 2009, p. 51–66.

[84] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: a fast address sanity checker,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference (ATC’12)*, 2012.

[85] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, “PAriCheck: an efficient pointer arithmetic checker for c programs,” in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS’10)*, 2010, p. 145–156.

[86] Y. Oiwa, “Implementation of the memory-safe full ANSI-C compiler,” in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI’09)*, 2009, p. 259–269.

[87] W. Xu, D. C. DuVarney, and R. Sekar, “An efficient and backwards-compatible transformation to ensure memory safety of c programs,” in *Proceedings of the 12th ACM SIGSOFT international symposium on Foundations of software engineering*, vol. 29, Oct. 2004, p. 117–126.

[88] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Softw. Pract. Exper.*, vol. 27, no. 1, p. 87–110, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199701)27:1<87::AID-SPE78>3.0.CO;2-P

[89] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09)*, 2009, p. 245–258.

[90] S. Nagarakatte, J. Zhao, M. M. Martin, and Steve Zdancewic, "CETS: compiler enforced temporal safety for c," in *Proceedings of the 2010 international symposium on Memory management (ISMM'10)*, June 2010, p. 31–40.

[91] M. Simpson and R. Barua, "MemSafe: ensuring the spatial and temporal memory safety of c at runtime," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, 2010, pp. 199–208.

[92] M. S. Simpson and R. K. Barua, "MemSafe: ensuring the spatial and temporal memory safety of c at runtime," *Softw. Pract. Exper.*, vol. 43, no. 1, p. 93–128, Jan. 2013.

[93] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'08)*, Mar. 2008, p. 103–114.

[94] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*, June 2012, p. 189–200.

[95] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (ASE'07)*, 2007, p. 284–292.

[96] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and efficient memory protection using dynamic tainting," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 87 –100, Jan. 2012.

[97] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman, "Architectural support for low overhead detection of memory violations," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, p. 652–657.

[98] W. Chuang, S. Narayanasamy, and B. Calder, "Accelerating meta data checks for software correctness and security," *Journal of Instruction -Level Parallism*, vol. 9, 2007.

[99] F. Qin, S. Lu, and Y. Zhou, "SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *11th International Symposium on High-Performance Computer Architecture (HPCA'11)*, 2005, pp. 291–302.

[100] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: efficient and programmable support for memory access monitoring and debugging," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*, 2007, p. 273–284.

[101] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "MemTracker: an accelerator for memory debugging and monitoring," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, p. 5:1–5:33, July 2009. [Online]. Available: http://doi.acm.org/10.1145/1543753.1543754

[102] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song, "Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP'13)*, San Francisco, CA, 2013.

[103] Chris Sanders, "Analyzing DLL hijacking attacks," 2010. [Online]. Available: http://www.windowsecurity.com/articles-tutorials/windows_os_security/Analyzing-DLL-Hijacking-Attacks.html

[104] "CVE-2011-1658," 2011. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1658

[105] "CVE-2012-0056," 2012. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0056

[106] F. L. Sang, V. Nicomette, and Y. Deswarte, "I/O attacks in intel PC-based architectures and countermeasures," in *SysSec Workshop (SysSec), 2011 First*. IEEE, July 2011, pp. 19–26.

[107] "CVE-2012-0217," 2012. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217

[108] J. Wei and C. Pu, "TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.

[109] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th conference on USENIX Security Symposium*, ser. SSYM'05, 2005, pp. 177–192.

[110] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*. New York, NY, USA: ACM, 2007, p. 552–561.

[111] "CVE-2003-0015," 2003. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0015

[112] "Beginners guide to use after free exploits IE6 0-day exploit development," 2012. [Online]. Available: http://www.garage4hackers.com/content/143-beginners-guide-use-after-free-exploits-ie-6-0-day-exploit-development.html

[113] J. Vanegue, "Zero-sized heap allocations vulnerability analysis," in *Proceedings of the 4th USENIX conference on Offensive technologies (WOOT'10)*, Aug. 2010, pp. 1–8.

[114] "CVE-2012-4681," 2012. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=2012-4681

[115] M. Payer and T. R. Gross, "String oriented programming: when ASLR is not enough," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*, 2013, p. 2:1–2:9.

[116] G. Argyros and A. Kiayias, "I forgot your password: randomness attacks against PHP applications," in *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[117] J. Manger, "A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'01)*. London, UK, UK: Springer-Verlag, 2001, p. 230–238.

[118] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP'12)*, May 2012, pp. 143 –157.

[119] "CWE-201: information exposure through sent data," 2008. [Online]. Available: http://cwe.mitre.org/data/definitions/201.html

[120] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[121] Sebastian Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," Tech. Rep., Sep. 2005.

[122] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," in *ACM Trans. Info. & Sys. Security*, 2012.

[123] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection (RAID'11).* Berlin, Heidelberg: Springer-Verlag, 2011, p. 121–141.

[124] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: exploit hardening made easy," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[125] Martin Abadi, M. Budiu, U. Erlingsson, and Jay Ligatti, "Control-flow integrity principles, implementations, and applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, 2005.

[126] "CVE-2011-1745," 2011. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1745

[127] Matthew Conover, "Double free vulnerabilities," 2007. [Online]. Available: http://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-1

[128] "CVE-2012-4792," 2012. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4792

[129] Solar Designer, "Linux kernel patch to remove stack exec permission," 1997. [Online]. Available: https://lkml.org/lkml/1997/4/12/7

[130] David LeBlanc, "Integer handling with the c++ SafeInt class," 2004. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

[131] Robert C. Seacord, *The CERT C Secure Coding Standard*, 1st ed. Addison-Wesley Professional, Oct. 2008.

[132] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, p. 203–216, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/173668.168635

[133] S. McCamant and G. Morrisett, “Evaluating SFI for a CISC architecture,” in *Proceedings of the 15th conference on USENIX Security Symposium (SS’06)*, 2006.

[134] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP’09)*, Oakland, CA, 2009, pp. 79–93.

[135] M. Payer, T. Hartmann, and T. R. Gross, “Safe loading - a foundation for secure execution of untrusted programs,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP’12)*, 2012, p. 18–32.

[136] A. Slowinska and H. Bos, “Pointless tainting? evaluating the practicality of pointer tainting,” in *ACM European conference on Computer systems*, 2009.

[137] M. Dalton, H. Kannan, and C. Kozyrakis, “Tainting is not pointless,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 88–92, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773933

[138] A. Slowinska and H. Bos, “Pointer tainting still pointless: (but we all see the point of tainting),” *ACM SIGOPS Operating Systems*, vol. 44, no. 3, p. 88–92, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1842733.1842748

[139] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, “Defeating memory corruption attacks via pointer taintedness detection,” in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN’05)*, 2005, p. 378–387.

[140] W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang, “InfoShield: a security architecture for protecting information usage in memory,” in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA’06)*, 2006, pp. 222–231.

[141] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: intermediate language and tools for analysis and transformation of c programs,” in *Proceedings of the 11th International Conference on Compiler Construction (CC’02)*. London, UK, UK: Springer-Verlag, 2002, p. 213–228.

[142] Wind River, “Wind river simics.” [Online]. Available: http://www.windriver.com/products/simics/

[143] Aeroflex Gaisler AB, “Leon3 processor.” [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon3

[144] ARM Ltd., “AMBA open specifications.” [Online]. Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[145] Xilinx Inc., “Virtex-5 FXT ML510 embedded development platform.” [Online]. Available: http://www.xilinx.com/products/boards-and-kits/HW-V5-ML510-G.htm

[146] National Institute of Standards and Technology, “Juliet test suite for C/C++.” [Online]. Available: http://samate.nist.gov/SRD/testsuite.php

[147] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI’07)*, June 2007, p. 89–100.

[148] M. C. Carlisle, “Olden: parallelizing programs with dynamic data structures on distributed-memory machines,” Ph.D. dissertation, Princeton University, Princeton, NJ, USA, 1996, UMI Order No. GAX96-27387.

[149] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO’04)*. Washington, DC, USA: IEEE Computer Society, 2004.

[150] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha, “Architectural support for safe software execution on embedded processors,” in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS’06)*, 2006, p. 106–111.

[151] Synopsys, Inc., “Synopsis DC ultra.” [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DCUltra/Pages/default.aspx

[152] S. H. Yong and S. Horwitz, "Protecting c programs from attacks via invalid pointer dereferences," *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, vol. 28, no. 5, p. 307–316, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/949952.940113

[153] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008, p. 377–388.

[154] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, p. 13–25, June 1997. [Online]. Available: http://doi.acm.org/10.1145/268806.268810

[155] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," 2005. [Online]. Available: http://sesc.sourceforge.net

[156] David Levinthal, "Cycle accounting analysis on intel core 2 processors," Intel Corporation, Tech. Rep. [Online]. Available: http://software.intel.com/sites/products/collateral/hpc/vtune/cycle_accounting_analysis.pdf

[157] Anand Lal Shimpi, "The clarkdale review: Intel's core i5 661, i3 540 & i3 530," 2010. [Online]. Available: http://www.anandtech.com/show/2901/2

[158] T. K. Prakash and L. Peng, "Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor," *ISAST Trans. Comput. Softw. Eng*, vol. 2, no. 1, pp. 36–41, 2008.

[159] M. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Experience with safe manual memory-management in cyclone," in *Proceedings of the 4th international symposium on Memory management (ISMM'04)*, 2004, p. 73–84.

[160] B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell, "Baggy bounds with accurate checking," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2012, pp. 195–200.

[161] "SPEC benchmarks." [Online]. Available: http://www.spec.org/

[162] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Discovering application-level insider attacks using symbolic execution," in *Emerging Challenges for Security, Privacy and Trust*, D. Gritzalis and J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 297, pp. 63–75. [Online]. Available: http://dblp.uni-trier.de/rec/bibtex/conf/sec/PattabiramanNKI09

[163] Cadence Design Systems, Inc., "Xtensa customizable processors." [Online]. Available: http://www.tensilica.com/products/xtensa-customizable